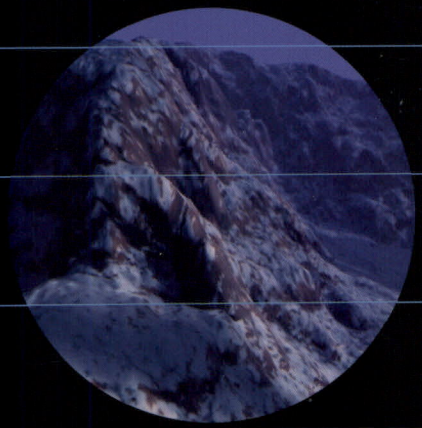
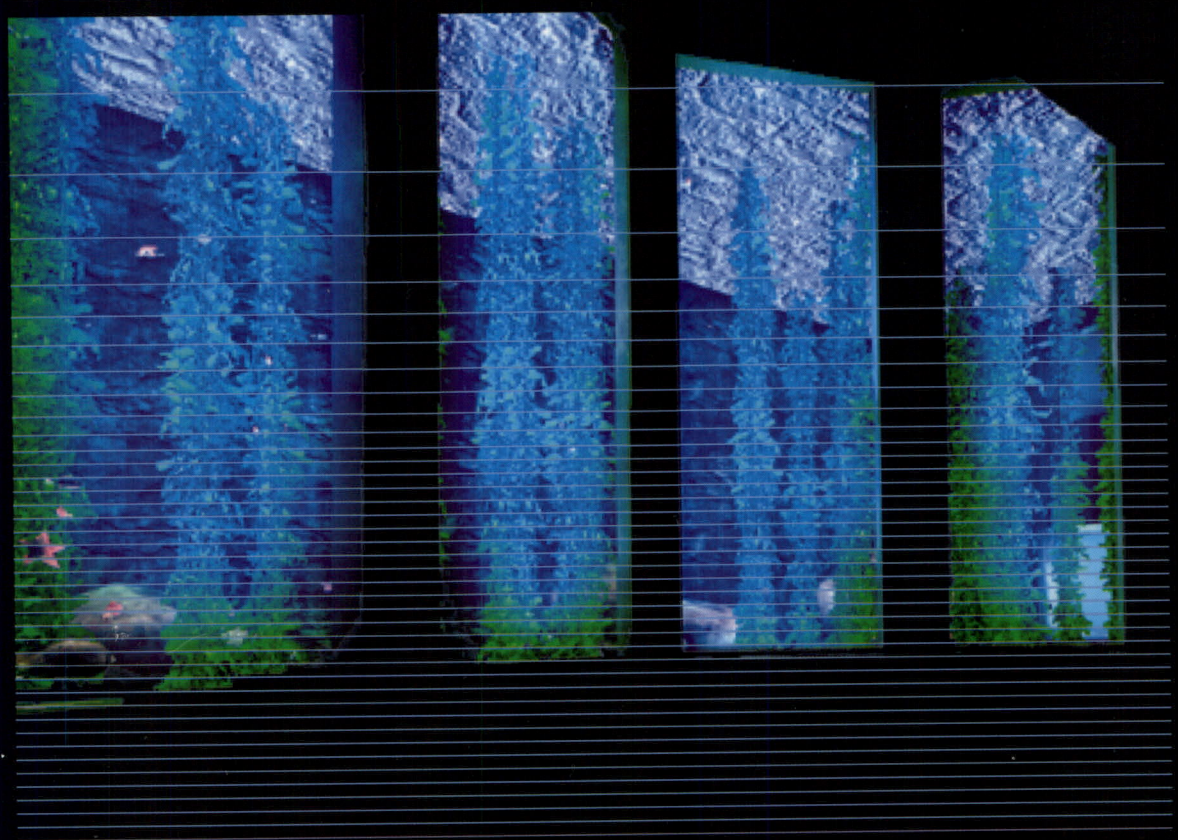
Volume 1 Issue 1 March 2004

JOGD Journal Of Game Development





ISSN: 1543-9399 (Individual)
ISSN: 1543-9402 (Library/Corporate)

CHARLES RIVER MEDIA, INC.
10 Downer Avenue
Hingham, Massachusetts 02043
781-740-0400
781-740-8816 (FAX)
info@charlesriver.com
www.charlesriver.com
www.jogd.com

JOURNAL OF GAME DEVELOPMENT

Publisher

CHARLES RIVER MEDIA, INC.
10 Downer Avenue
Hingham, Massachusetts 02043

Volume I, Issue I

LETTER FROM THE EDITOR

Welcome to the first issue of the *Journal of Game Development*. I'd like to start by thanking Charles River Media for recognizing the need for a peer-reviewed journal focusing on technical research relating to game development. There have not been enough publication venues for game-related research, and I'm excited to be able to help remedy the issue. Charles River Media has a great deal of experience in technical publications, including the *Game Programming Gems* series, and I've already benefited from their experience.

The next order of business is to introduce myself to you. I'm a research scientist at the University of Southern California's Institute for Creative Technologies. In my research, I focus on artificial intelligence techniques that apply to video games and military training simulations. Two specific areas that I'm currently investigating are adaptive strategic planning for real-time strategy games and NPCs who can explain their behavior to the user. I received a bachelor's degree from Williams College, a master's degree of science from the University of Tennessee, and a Ph.D. from the University of Michigan—representing 13 straight years of school. Although I have never worked as a commercial game developer, I've spent a great deal of time working with developers, and have, in small ways, participated in the development of a few games.

The central focus of each issue of the *Journal of Game Development* will be four peer-reviewed technical articles. In addition, each issue will include an invited article that is less technical in nature and presents a different perspective on games. We have some other ideas for future issues, so keep an eye out for future developments. As the editor-in-chief of the *Journal of Game Development*, my main job is to make sure that each article is valuable to the readers. *JOGD* targets both commercial developers and academic researchers, which makes this a challenge. If you have any suggestions about how we can improve the journal, please send me an e-mail at *editorinchief@jogd.com*.

Michael van Lent
March 2004

INTRODUCTION

Welcome to the *Journal of Game Development* (*JOGD*). *JOGD* has been created as a publication venue for technical research papers that have relevance to the development of video games and other forms of interactive entertainment. The video game industry is a large and rapidly evolving business driven by the intelligence and hard work of game designers and programmers. In building today's games, these developers are encountering many of the problems that have been the subject of years, sometimes decades, of academic research, and are uncovering new problems that could challenge researchers for years to come. Despite this, there hasn't been nearly enough communication between the commercial game industry and the academic research community. Both communities need to make an effort to become familiar with the other. We hope that *JOGD* will serve as one channel of communication.

Game developers will find that *JOGD* contains articles on the latest ideas and technologies being explored by other developers, and faculty and students from some of the finest universities in the United States and abroad. As consumers become more demanding and games become more complex, game developers will find that they need new technologies and ideas to stay competitive. The research community represents a largely untapped source. Developers who have reached out to academic researchers have been surprised to find avid gamers who have chosen their research fields with the goal of making future games more entertaining. Some of the research ideas are ready to be used in today's games. Some of the ideas won't be ready until next year or next decade. Some of the research will result in new genres of games or even change our perception of what games are.

The research community will find that *JOGD* contains both valuable information about the investigations of their peers and information about the challenges faced by commercial game developers and the solutions they have developed. Games represent an important commercial application of a number of technical fields such as real-time computer graphics, computer networking, and artificial intelligence. Researchers working in any field related to the commercial game industry must remain informed about the challenges faced by game developers and the novel solutions they have developed. An experienced game programmer is as likely to have good ideas as a tenured faculty member. Academic researchers who have reached out to game developers have found smart people who are thinking about very interesting problems and often have some familiarity with the research literature.

To support communication and collaboration between these two communities, the *Journal of Game Development* encourages submissions from both academic researchers and commercial developers. Research topics suitable for publication in *JOGD* include, but are not limited to, artificial intelligence, computer graphics, animation, physics simulation, networking, digital audio, and modeling and simulation. *JOGD* will be published quarterly, and each issue will contain four peer-reviewed articles. Articles will include basic and applied research papers and survey articles summarizing a well-established field of research. Work published in *JOGD* will be relevant to academic researchers and/or commercial developers and will describe a novel technical advance not previously published. Submissions will be judged on relevance, novelty of the work, technical soundness, and clarity of the writing.

Please enjoy this issue of the *Journal of Game Development* and consider submitting an idea of your own. *JOGD* will only be as good as the ideas of our authors. MvL.

Near Optimal Hierarchical Path-Finding

Adi Botea, Martin Müller, Jonathan Schaeffer
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
adib@cs.ualberta.ca
mmueller@cs.ualberta.ca
jonathan@cs.ualberta.ca

Abstract

The problem of path-finding in commercial computer games has to be solved in real time, often under constraints of limited memory and CPU resources. The computational effort required to find a path, using a search algorithm such as A*, increases with size of the search space. Hence, pathfinding on large maps can result in serious performance bottlenecks.

This article presents HPA* (Hierarchical Path-Finding A*), a hierarchical approach for reducing problem complexity in path-finding on grid-based maps. This technique abstracts a map into linked local clusters. At the local level, the optimal distances for crossing each cluster are precomputed and cached. At the global level, clusters are traversed in a single big step. A hierarchy can be extended to more than two levels. Small clusters are grouped together to form larger clusters. Computing crossing distances for a large cluster uses distances computed for the smaller contained clusters.

Our method is automatic and does not depend on a specific topology. Both random and real-game maps are successfully handled using no domain-specific knowledge. Our problem decomposition approach works very well in domains with a dynamically changing environment. The technique also has the advantage of simplicity and is easy to implement. If desired, more sophisticated, domain-specific algorithms can be plugged in for increased performance.

The experimental results show a great reduction of the search effort. Compared to a highly optimized A*, HPA* is shown to be up to 10 times faster, while finding paths that are within 1 percent of optimal.

INTRODUCTION

The problem of path-finding in commercial computer games has to be solved in real time, often under constraints of limited memory and CPU resources. Hierarchical search is acknowledged as an effective approach to reduce the complexity of this problem. However, no detailed study of hierarchical path-finding in commercial games has been published. Part of the explanation is that game companies usually do not make their ideas and source code available.

The industry standard is to use A* [Stout 1996] or iterative-deepening A*, IDA* [Korf 1985]. A* is generally faster, but IDA* uses less memory. There are numerous enhancements to these algorithms to make them run faster or explore a smaller search tree. For many applications, especially those with multiple moving NPCs (such as in real-time strategy games), these time and/or space requirements are limiting factors.

In this article we describe HPA*, a new method for hierarchical path-finding on grid-based maps, and present performance tests. Our technique abstracts a map into linked local clusters. At the local level, the optimal distances for crossing the cluster are precomputed and cached. At the global level, an action is to cross a cluster in a single step rather than moving to an adjacent atomic location.

Our method is simple, easy to implement, and generic, as we use no application-specific knowledge and apply the technique independently of the map properties. We handle variable cost terrains and various topology types such as forests, open areas with obstacles of any shape, or building interiors—without any implementation changes.

For many real-time path-finding applications, the complete path is not needed. Knowing the first few moves of a valid path often suffices, allowing a mobile unit to start moving in the right direction. Subsequent events may result in the unit having to change its plan, obviating the need for the rest of the path. A* returns a complete path. In contrast, HPA* returns a complete path of subproblems. The first subproblem can be solved, giving a unit the first few moves along the path. As needed, subsequent subproblems can be solved providing additional moves. The advantage here is that if the unit has to change its plan, then no effort has been wasted on computing a path to a goal node that was never needed.

The hierarchical framework is suitable for static and dynamically changing environments. In the latter case, first assume that local changes can occur on immobile topology elements (e.g., a bomb destroys a bridge). We recompute the information extracted from the modified cluster locally and keep the rest of the framework unchanged. Second, assume that there are many mobile units on the map, and a computed path can become blocked by another unit. We compute an abstract path with reduced effort and do not spend additional effort to refine it to the low-level representation. We quickly get the character moving in a proven good direction and refine parts of the abstract path as the character needs them. If the path becomes blocked, we replan for another abstract path from the current position of the character.

The hierarchy of our method can have any number of levels, making it scalable for large problem spaces. When the problem map is large, a larger number of levels can be the answer for reducing the search effort, for the price of more storage and pre-processing time.

Our technique produces suboptimal solutions, trading optimality for improved execution performance. After applying a path-smoothing procedure, our solutions are within 1 percent of optimal.

Motivation

Consider the problem of traveling by car from Los Angeles, California, to Toronto, Ontario. Specifically, what is the minimum distance to travel by car from 1234 Santa Monica Boulevard in Los Angles to 4321 Yonge Street in Toronto? Given a detailed road map of North America, showing *all* roads annotated with driving distances, an A* implementation can compute the optimal (minimum distance) travel route. This might be an expensive computation, given the sheer size of the road map.

Of course, a human travel planner would never work at such a low level of detail. They would solve three problems:

1. Travel from 1234 Santa Monica Boulevard to a major highway leading out of Los Angeles.
2. Plan a route from Los Angeles to Toronto.
3. Travel from the incoming highway in Toronto to 4321 Yonge Street.

The first and third steps would require a detailed road map of each city. Step 2 could be done with a high-level map, with roads connecting cities, abstracting away all the detail within the city. In effect, the human travel planner uses abstraction to find a quick route from Los Angles to Toronto. However, by treating cities as black boxes, this search is not guaranteed to find the shortest route. For example, although it may be faster to stay on a highway, for some cities where the highway goes around the city, leaving the highway and going through the city might be a shorter route. Of course, it may not be a faster route (city speeds are slower than highway speeds), but in this example we are trying to minimize travel distance.

Abstraction could be taken to a higher level: do the planning at the state/province level. Once the path reaches a state boundary, compute the best route from state to state. Once you know your entrances and exits from the states, then plan the interstate routes. Again, this will work but may result in a suboptimal solution.

Taken to the extreme, the abstraction could be at the country level: travel from the United States to Canada. Clearly, there comes a point where the abstraction becomes so coarse it is effectively useless.

We want to adopt a similar abstraction strategy for computer game path-finding. We could use A* on a complete 1000×1000 map, but that represents a potentially huge search space. Abstraction can be used to reduce this dramatically. Consider each 10×10 block of the map as being a "city." Now we can search in a map of 100×100 cities. For each city, we know the city entrances and the costs of crossing the city for all the entrance pairs. We also know how to travel between cities. The problem then reduces to three steps:

1. Start node: Within the block containing the start node, find the optimal path to the borders of the block.
2. Search at the block level (100×100 blocks) for the optimal path from the block containing the start node to the block containing the goal node.
3. Goal node: Within the block containing the goal node, find the optimal path from the border of the block to the goal.

The result is a much faster search giving nearly optimal solutions. Further, the abstraction is topology independent; there is no need for a level designer to break the grid manually into high-level features or annotate it with way-points.

Contributions

The contributions of this article include:

- HPA*, a new hierarchical path-finding algorithm (including pseudocode and source code) that is domain independent and works well for static and dynamic terrain topologies.
- Experimental results for hierarchical search on a variety of game mazes (from BioWare's *Baldur's Gate*, showing up to a ten-fold speed improvement in exchange for a 1 percent degradation in path quality.
- Variations on the hierarchical search idea appear to be in use by several game companies, although most of their algorithmic details are not public. To the best of our knowledge, this is the first scientific study of using hierarchical A* in the domain of commercial computer games.

The next section contains a brief overview of the background literature. Subsequent sections present our new approach to hierarchical A* and an evaluation of its performance. We then present our conclusions and topics for further research, along with the pseudocode for our algorithm.

Literature Review

The first part of this section summarizes hierarchical approaches used for path-finding in commercial games. The second part reviews related work in a more general context, including applications to other grid domains such as robotics.

Path-finding using a two-level hierarchy is described in [Rabin 2000]. The author provides only a high-level presentation of the approach. The problem map is abstracted into clusters such as rooms in a building or square blocks on a field. An abstract action crosses a room from the middle of an entrance to another. This method has similarities to our work. First, both approaches partition the problem map into clusters such as square blocks. Second, abstract actions are block crossings (as opposed to going from one block center to another block center). Third, both techniques abstract a block entrance into one transition point (in fact, we allow either one or two points). This leads to fast computation but gives up the solution optimality. There are also significant differences between the two approaches. We extend our hierarchy to several abstraction levels and do this abstraction in a domain-independent way. We also precompute and cache optimal distances for block crossing, reducing the costs of the online computation.

Another important hierarchical approach for path-finding in commercial games uses *points of visibility* [Rabin 2000]. This method exploits the domain local topology to define an abstract graph that covers the map efficiently. The graph nodes represent the corners of convex obstacles. For each node, edges are added to all the nodes that can be seen from the current node (i.e., they can be connected with a straight line).

This method provides solutions of good quality. It is particularly useful when the number of obstacles is relatively small and they have a convex polygonal shape (e.g., building interiors). The efficiency of the method decreases when many obstacles are present and/or their shape is not a convex polygon. Consider the case of a map containing a forest, which is a dense collection of small-sized obstacles. Modeling such a topology with points of visibility would result in a large graph (in terms of number of nodes and edges) with short edges. Therefore, the key idea of traveling long distances in a single step wouldn't be efficiently exploited. When the problem map contains concave or curved shapes, the method either has poor performance or needs sophisticated engineering to build the graph efficiently. In fact, the need for algorithmic or designer assistance to create the graph is one of the disadvantages of the method. In contrast, our approach works for many kinds of maps and does not require complex domain analysis to perform the abstraction.

The *navigation mesh* (NavMesh) is a powerful abstraction technique useful for 2D and 3D maps. In a 2D environment, this approach covers the unblocked area of a map with a minimal set of convex polygons. A method for building a near-optimal NavMesh is presented in [Tozour 2002]. This method relaxes the condition of the minimal set of polygons and builds a map coverage much faster.

Besides commercial computer games, path-finding has applications in many research areas. Path-finding approaches based on topological abstraction that have been explored in robotics domains are especially relevant for the work described in this article. *Quadtrees* [Samet 1988] have been proposed as a way of doing hierarchical map decomposition. This method partitions a map into square blocks with different sizes so that a block contains either only walkable cells or only blocked cells. The problem map is initially partitioned into four blocks. If a block contains both obstacle cells and walkable cells, then it is further decomposed into four smaller blocks, and so on. An action in this abstracted framework is to travel between the centers of two adjacent blocks. Since the agent always goes to the middle of a box, this method produces suboptimal solutions.

To improve the solution quality, quadtrees can be extended to *framed quadtrees* [Chen 1995; Yahja 1998]. In framed quadtrees, the border of a block is augmented with cells at the highest resolution. An action crosses a block between any two border cells. Since this representation permits many angles of direction, the solution quality improves significantly. However, framed quadtrees use more memory than quadtrees.

Framed quadtrees are more similar to our work than quadtrees, since we also use block crossings as abstract actions. However, we don't consider *all* the cells on the block border as entrance points. We reduce the number of block entrance points by abstracting an entrance into one or two such points. Moreover, our approach allows blocks to contain obstacles. This means that the distance between two transition points is not necessarily linear. For this reason, we have to compute optimal paths between entrance points placed on the border of the same block.

A multilevel hierarchy has been used to enhance the performance of multiple goal path-planning in a Markov Decision Process (MDP) framework [Moore 1999]. The problem posed is to learn efficiently near optimal policies $\pi^*(x,y)$ to travel from x to y for all pairs (x,y) of map locations. The number of policies that have to be computed and stored is quadratic in the number of map cells. To improve both the memory and time requirements (for the price of losing optimality), a multilevel structure is used—a so called *airport hierarchy*. All locations on the problem map are airports that are assigned to different hierarchical levels. The strategy for traveling from x to y is similar to traveling by plane in the real world. First, travel to bigger and bigger airports until we reach an airport that is big enough to have a connection to the area that contains the destination. Second, go down in the

hierarchy by traveling to smaller airports until the destination is reached. This approach is very similar to the strategy outlined earlier.

An analysis of the nature of path-finding in various frameworks is performed in [Reese 1999]. The authors classify path-finding problems based on the type of the results that are sought, the environment type, the amount of information available, and so forth. Challenges specific to each problem type and solving strategies such as re-planning and using dynamic data structures are briefly discussed.

A hierarchical approach for shortest path algorithms that has similarities with HPA* is analyzed in [Shekhar 1997]. This work decomposes an initial problem graph into a set of fragment subgraphs and a global boundary subgraph that links the fragment subgraphs. Shortest paths are computed and cached for future use, similarly to the caching that HPA* performs for cluster traversal routes. The authors analyze what shortest paths (i.e., from which subgraphs) to cache, and what information to keep (i.e., either complete path or only cost) for best performance when limited memory is available.

Another technique related to HPA* is Hierarchical A* [Holte 1996], which also uses hierarchical representations of a space with the goal of reducing the overall search effort. However, the way in which hierarchical representations are used is different in these two techniques. While our approach uses abstraction to structure and enhance the representation of the search space, Hierarchical A* is a method for automatically generating domain-independent heuristic state evaluations. In single-agent search, a heuristic function that evaluates the distance from a state to the goal is used to guide the search process. The quality of such a function greatly affects the quality of the whole search algorithm. Starting from the initial space, Hierarchical A* builds a hierarchy of abstract spaces until an abstract one-state space is obtained. When building the next abstract space, several states of the current space are grouped to form one abstract state in the next space. In this hierarchy, an abstract space is used to compute a heuristic function for the previous space.

HIERARCHICAL PATH-FINDING

Our hierarchical approach implements the strategy described previously. Searching for an abstract solution in our hierarchical framework is a three-step process called *online search*. First, travel to the border of the neighborhood that contains the start location. Second, search for a path from the border of the start neighborhood to the border of the goal neighborhood. This is done on an abstract level, where search is simpler and faster. An action travels across a relatively large area, with no need to deal with the details of that area. Third, complete the path by traveling from the border of the goal neighborhood to the goal position.

The abstracted graph for online search is built using information extracted from the problem maze. We discuss in more detail how the framework for hierarchical search is built (preprocessing) and how it is used for path finding (online search). Initially, we focus on building a hierarchy of two levels: one low level and one abstract level. Adding more hierarchical levels is discussed at the end of this section. We illustrate how our approach works on the small 40×40 map shown in Figure 1.1a.

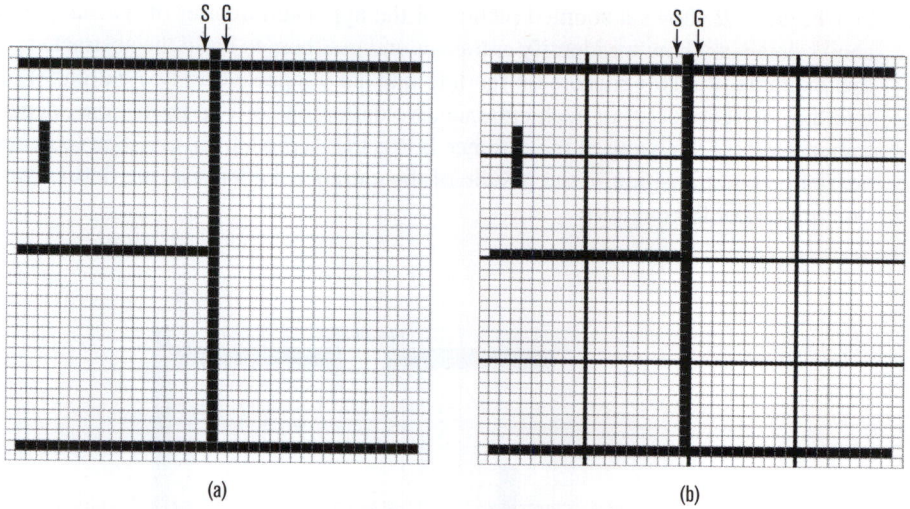


FIGURE 1.1 (a) The 40 × 40 maze used in our example. The obstacles are painted in black. *S* and *G* are the start and the goal nodes. (b) The bold lines show the boundaries of the 10 × 10 clusters.

Pre-Processing a Grid

The first step in building the framework for hierarchical search defines a topological abstraction of the maze. We use this maze abstraction to build an abstract graph for hierarchical search.

The topological abstraction covers the maze with a set of disjunct rectangular areas called *clusters*. The bold lines in Figure 1.1b show the abstract clusters used for topological abstraction. In this example, the 40 × 40 grid is grouped into 16 clusters of size 10 × 10. Note that no domain knowledge is used to do this abstraction (other than, perhaps, tuning the size of the clusters).

For each border line between two adjacent clusters, we identify a (possibly empty) set of entrances connecting them. An entrance is a maximal obstacle-free segment along the common border of two adjacent clusters c_1 and c_2, formally defined as shown next. Consider the two adjacent lines of tiles l_1 and l_2, one in each cluster, that determine the border edge between c_1 and c_2. For a tile $t \in l_1 \cup l_2$, we define symm(t) as being the symmetrical tile of t with respect to the border between c_1 and c_2. Note that t and symm(t) are adjacent and never belong to the same cluster. An entrance e is a set of tiles that respects the following conditions:

The border limitation condition: $e \subset l_1 \cup l_2$. This condition states that an entrance is defined along and cannot exceed the border between two adjacent clusters.

The symmetry condition: $\forall t \in l_1 \cup l_2 : t \in e \Leftrightarrow \text{symm}(t) \in e$.

The obstacle free condition: An entrance contains no obstacle tiles.

The maximality condition: An entrance is extended in both directions as long as the previous conditions remain true.

Figure 1.2 shows a zoomed picture of the upper-left quarter of the sample map. The picture shows details on how we identify entrances and use them to build the abstracted problem graph. In this example, the two clusters on the left side are connected by two entrances of width 3 and of width 6, respectively. For each entrance, we define one or two *transitions*, depending on the entrance width. If the width of the entrance is less than a predefined constant (6 in our example), then we define one transition in the middle of the entrance. Otherwise, we define two transitions, one on each end of the entrance.

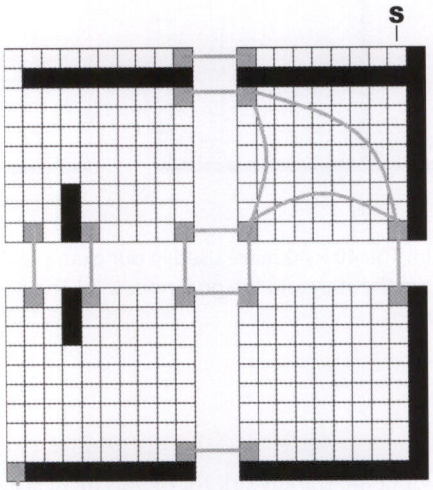
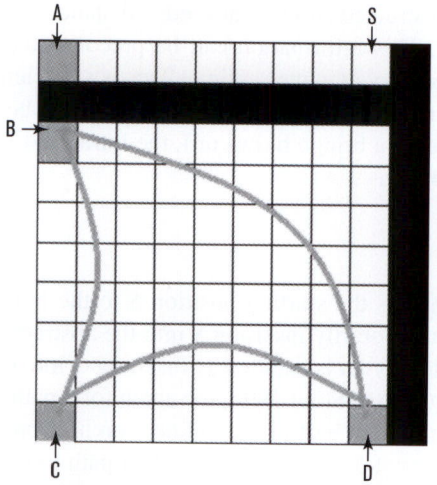


FIGURE 1.2 Abstracting the top-left corner of the maze. All abstract nodes and inter-edges are shown in light gray. For simplicity, intra-edges are shown only for the top-right cluster.

We use transitions to build the abstract problem graph. For each transition, we define two nodes in the abstract graph and an edge that links them. Since such an edge represents a transition between two clusters, we call it an *inter-edge*. Inter-edges always have length 1. For each pair of nodes inside a cluster, we define an edge linking them, called an *intra-edge*. We compute the length of an intra-edge by searching for an optimal path inside the cluster area.

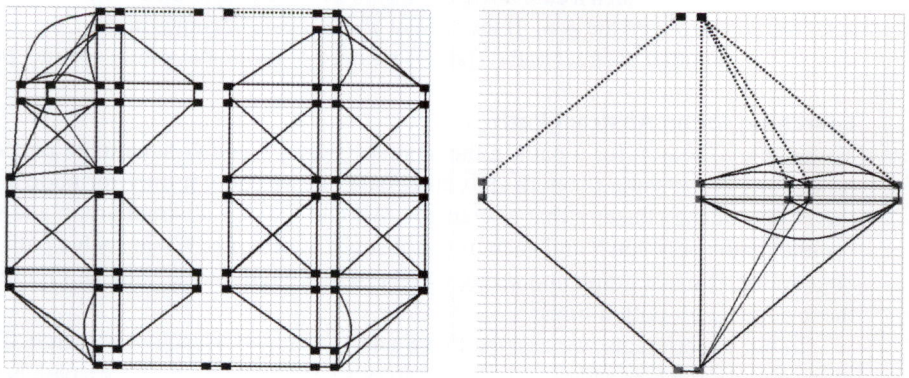
Figure 1.2 shows all the nodes (light-gray-squares), all the inter-edges (light-gray-lines), and part of the intra-edges (for the top-right cluster). Figure 1.3 shows the details of the abstracted internal topology of the cluster in the top-right corner of Figure 1.2. The data structure contains a set of nodes as well as distances between them. We define the distance as 1 for a straight transition and 1.42 for a diagonal transition. (The generic path-finding library that we used in our experiments utilizes this value for approximating $\sqrt{2}$. A slightly more appropriate approximation would probably be 1.41.) We only cache distances between nodes and discard the actual optimal paths corresponding to these distances. If desired, the paths can also be stored, for the price of more memory usage. See the section *Path Refinement* for a discussion.



	A	B	C	D
A		∞	∞	∞
B			7.00	10.94
C				8.00
D				

FIGURE 1.3 Cluster-internal path information.

Figure 1.4a shows the abstract graph for our running example. The picture includes the result of inserting the start and goal nodes *S* and *G* into the graph (the dotted lines), which is described in the next subsection. The graph has 68 nodes, including *S* and *G*, which can change for each search. At this level of abstraction, there are 16 clusters with 43 inter-connections and 88 intra-connections. There are two additional edges that link *S* and *G* to the rest of the graph. For comparison, the low-level (nonabstracted) graph contains 1,463 nodes, one for each unblocked tile, and 2,714 edges.



(a) (b)

FIGURE 1.4 (a) The abstract problem graph in the hierarchy with one low level and one abstract level. (b) Level 2 of the abstract graph in the 3-level hierarchy. The dotted edges connect *S* and *G* to the rest of each graph.

Once the abstract graph has been constructed and the intra-edge distances computed, the grid is ready to use in a hierarchical search. This information can be precomputed (before a game ships), stored on disk, and loaded into memory at game runtime. This is sufficient for static (non-changing) grids. For dynamically changing grids, the precomputed data has to be modified at runtime. When the grid topology changes (e.g., a bridge blows up), the intra- and inter-edges of the affected local clusters need to be recomputed.

Online Search

The first phase of the online search connects the starting position S to the border of the cluster containing S. This step is completed by temporarily inserting S into the abstract graph. Similarly, connecting the goal position G to its cluster border is handled by inserting G into the abstract graph.

After S and G have been added, we use A* [Stout 1996] to search for a path between S and G in the abstract graph. This is the most important part of the online search. It provides an abstract path, the actual moves from S to the border of S's cluster, the abstract path to G's cluster, and the actual moves from the border of G's cluster to G.

The last two steps of the online search are optional:

1. Path refinement can be used to convert an abstract path into a sequence of moves on the original grid.
2. Path-smoothing can be used to improve the quality of the path-refinement solution.

The abstract path can be refined in a post-processing step to obtain a detailed path from S to G. For many real-time path-finding applications, the complete path is not needed—only the first few moves. This information allows the character to start moving in the right direction toward the goal. In contrast, A* must complete its search and generate the entire path from S to G before it can determine the first steps of a character.

Consider a domain where dynamic changes occur frequently (e.g., there are many mobile units traveling around). In such a case, after finding an abstract path, we can refine it gradually as the character navigates toward the goal. If the current abstract path becomes invalid, the agent discards it and searches for another abstract path. There is no need to refine the whole abstract path in advance.

Searching for an Abstract Path

To be able to search for a path in the abstract graph, S and G have to be part of the graph. The processing is the same for both start and goal, and we show it only for node S. We connect S to the border of the cluster c that contains it. We add S to the abstract graph and search locally for optimal paths between S and each of the abstract nodes of c. When such a path exists, we add an edge to the abstract graph and set its weight to the length of the path. In Figure 1.4 we represent these edges with dotted lines.

In our experiments, we assume that S and G change for each new search. Therefore, the cost of inserting S and G is added to the total cost of finding a solution. After a path is found, we remove S and G from the graph. However, in practice, this computation can be done more efficiently. Consider a game when many units have to find a path to the same goal. In this case, we insert G once and reuse it. The cost of inserting G is amortized over several searches. In general, a cache can be used to store connection information for popular start and goal nodes.

After inserting S and G, the abstract graph can be used to search for an abstract path between S and G. We run a standard single-agent search algorithm such as A* on the abstract graph.

Path Refinement

Path refinement translates an abstract path into a low-level path. Each cluster crossing in the abstract path is replaced by an equivalent sequence of low-level moves.

If the cluster pre-processing cached these move sequences attached to the intra-edges, then refinement is simply a table lookup. Otherwise, we perform small searches inside each cluster along the abstract path to rediscover the optimal local paths. Two factors limit the complexity of the refinement search. First, abstract solutions are guaranteed to be correct, provided that the environment does not change after finding an abstract path. This means that we never have to backtrack and replan for correcting the abstract solution. Second, the initial search problem has been decomposed into several very small searches (one for each cluster on the abstract path), with low complexity.

Path Smoothing

The topological abstraction phase defines only one transition point per entrance. While this is efficient, it gives up the optimality of the computed solutions. Solutions are optimal in the abstract graph but not necessarily in the initial problem graph.

To improve the solution quality (i.e., length and aesthetics), we perform a post-processing phase for path smoothing. Our technique for path smoothing is simple, but produces good results. The main idea is to replace local suboptimal parts of the solution by straight lines. We start from one end of the solution. For each node in the solution, we check whether we can reach a subsequent node in the path in a straight line. If this happens, then the linear path between the two nodes replaces the initial suboptimal sequence between these nodes.

Experimental Results for Example

The experimental results of our running example are summarized in the first two rows of Table 1.1. L-0 represents running A* on the low-level graph (we call this level 0). L-1 uses two hierarchy levels (level 0 and level 1), and L-2 uses three hierarchy levels (level 0, level 1, and level 2). The meaning of the last row, labeled L-2, is described later in this section.

TABLE 1.1 Summary of results of our running example. We show the number of expanded nodes. SG is the effort for inserting S and G into the graph. Abstract is the sum of the previous two columns. This measures the effort for finding an abstract solution. Refinement shows the effort for complete path refinement.

Search Technique	SG	Main	Abstract	Refinement
L-0	0	1,462	1,462	0
L-1	16	67	83	145
L-2	41	7	48	161

Low-level (original grid) search using Manhattan distance as the heuristic has poor performance. Our example has been chosen to show a worst-case scenario. Without abstraction, A* will visit all the unblocked positions in the maze. The search expands 1,462 nodes. The only factor that limits the search complexity is the maze size. A larger map with a similar topology represents a hard problem for A*.

The performance is greatly improved by using hierarchical search. When inserting S into the abstract graph, it can be linked to only one node on the border of the starting cluster. Therefore, we add one node (corresponding to S) and one edge that links S to the only accessible node in the cluster. Finding the edge uses a search that expands 8 nodes. Inserting G into the graph is identical.

A* is used on the abstracted graph to search for a path between S and G. Searching at level 1 also expands all the nodes of the abstract graph. The problem is also a worst-case scenario for searching at level 1. However, this time, the search effort is much reduced.

The main search expands 67 nodes. In addition, inserting S and G expands 16 nodes. In total, finding an abstract path requires 83 node expansions. This effort is enough to provide a solution for this problem—the moves from S to the edge of its cluster and the abstract path from the cluster edge to G. If desired, the abstract path can be refined, partially or completely, for additional cost. The worst case is when we have to refine the path completely and no actual paths for intra-edges were cached. For each intra-edge (i.e., cluster crossing) in the path, we perform a search to compute a corresponding low-level action sequence. There are 12 such small searches, which expand a total of 145 nodes.

Adding Levels of Hierarchy

The hierarchy can be extended to several levels, transforming the abstract graph into a *multilevel* graph. In a multilevel graph, nodes and edges have labels showing their level in the abstraction hierarchy. We perform path-finding using a combination of small searches in the graph at various abstraction levels. Additional levels in the hierarchy can reduce the search effort, especially for large mazes. Details on efficient searching in a multilevel graph are provided later in this article. To build a multilevel graph, we structure the maze abstraction on several levels. The higher the level, the larger the clusters in the maze decomposition. The clusters for level l are called l-clusters. We build each new level on top of the existing structure. Building the 1-clusters was presented previously. For $l \geq 2$, an l-cluster is obtained by grouping together $n \times n$ adjacent $(l-1)$-clusters, where n is a parameter.

Nodes on the border of a newly created l-cluster update their level to l (we call these l-nodes). Inter-edges that make transitions between l-clusters also increase their level to l (we call these l-inter-edges).

We add intra-edges with level l (i.e., l-intra-edges) for pairs of communicating l-nodes placed on the border of the same l-cluster. The weight of such an edge is the length of the shortest path that connects the two nodes within the cluster, using only $(l-1)$-nodes and $(l-1)$-edges. More details are provided later in this article.

Inserting S into the graph iteratively connects S to the nodes on the border of the l-cluster that contains it, with l increasing from 1 to the maximal abstraction level. Searching for a path between S and a l-node is restricted to level $l-1$ and to the area of the current l-cluster that contains S. We perform an identical processing for G.

The way we build the abstract graph ensures that we always find the same solution, no matter how many abstract levels we use. In particular, adding a new level $l \geq 2$ to the graph does not di-

minish the solution quality. Here we provide a brief intuitive explanation rather than a formal proof of this statement. A new edge added at level l corresponds to an existing shortest path at level $l-1$. The weight of the new edge is set to the cost of the corresponding path. Searching at level l finds the same solution as searching at level $l-1$, only faster.

In our example, adding an extra level with $n = 2$ creates four large clusters, one for each quarter of the map. The whole of Figure 1.2 is an example of a single 2-cluster. This cluster contains 2×2 1-clusters of size 10×10. Besides S, the only other 2-node of this cluster is the one in the bottom-left corner. Compared to level 1, the total number of nodes at the second abstraction level is reduced even more. Level 2, where the main search is performed, has 14 nodes (including S and G). Figure 1.4b shows level 2 of the abstract graph. The edges pictured as dotted lines connect S and G to the graph at level 2.

Abstraction level 2 is a good illustration of how the pre-processing solves local constraints and reduces the search complexity in the abstract graph. The 2-cluster shown in Figure 1.2 is large enough to contain the large dead end "room" that exists in the local topology. At level 2, we avoid any useless search in this "room" and go directly from S to the exit in the bottom-left corner.

After inserting S and G, we are ready to search for a path between S and G. We search only at the highest abstraction level. Since start and goal have the highest abstraction level, we will always find a solution, assuming that one exists. The result of this search is a sequence of nodes at the highest level of abstraction. If desired, the abstract path can repeatedly be refined until the low-level solution is obtained.

Experimental Results for Example with 3-Level Hierarchy

The third row of Table 1.1 shows numerical data for our running example with a 3-level hierarchy (i.e., with three levels: $L-0, L-1$, and $L-2$).

As discussed earlier, connecting S and G to the border of their 1-clusters expands 16 nodes in total. Similarly, we now connect S and G to the border of their 2-clusters. These searches at level 1 expand three nodes for S and 22 nodes for G.

The main search at level 2 expands only seven nodes. No nodes other than the ones in the abstract path are expanded. This is an important improvement, if we consider that the main search in the level 1 graph expanded all nodes in the graph. In total, finding an abstract solution in the extended hierarchy requires 48 nodes.

It is worth it to remark that, after adding a new abstraction level, the cost for inserting S and G dominates the main search cost. This illustrates the general characteristic of the method that the cost for inserting S and G increases with the number of levels, whereas the main search becomes simpler. Finding a good trade-off between these searches is important for optimizing the performance.

Table 1.1 also shows the costs for complete solution refinement. Refining the solution from level 2 to level 1 expands 16 nodes, and refining from level 1 to level 0 expands 145 nodes, for a total of 161 nodes.

Storage Analysis

Besides the computational speed, the amount of storage that a method uses for path-finding is another important performance indicator. Two main factors influence the amount of memory that our hierarchical approach uses: the size of the problem graph and the size of the open list used by A*. We discuss these two factors in more detail in the rest of this section. For the graph storage, we include both an empirical analysis and a worst-case theoretical discussion.

Graph Storage Requirements

Table 1.2 shows the average size of the problem graph for our *Baldur's Gate* test suite. Details about this data set and settings such as cluster sizes, or edge definition in the original problem graph are provided later in this article. We compare the original low-level graph to the abstract graphs in hierarchies with one, two, and three abstract levels (not counting level 0). In Table 1.2, we show the number of nodes N, the number of inter-edges E_1, and the number of intra-edges E_2. For the multi-level graphs, we show both the total numbers and the numbers for each level L_i, $i \in \{1,2,3\}$.

TABLE 1.2 The average size of the problem graph in *Baldur's Gate*. Graph 0 is the initial low-level graph. Graph 1 represents a graph with one abstract level (L_1), Graph 2 has two abstract levels (L_1, L_2), and Graph 3 has three abstract levels (L_1, L_2, L_3). N is the number of nodes, E_1 is the number of inter-edges, and E_2 is the number of intra-edges.

	Graph 0	Graph 1		Graph 2			Graph 3			
		L_1	Total	L_1	L_2	Total	L_1	L_2	L_3	Total
N	4,469	367	**367**	186	181	**367**	186	92	89	**367**
E_1	16,420	198	**198**	100	98	**198**	100	50	48	**198**
E_2	0	722	**722**	722	662	**1,384**	722	662	462	**1,846**

The data show that the storage overhead of the abstract graph is small compared to the size of the original problem graph. Adding a new graph level updates the level of some existing nodes and inter-edges without creating any new objects of these types. The only overhead consists of the new intra-edges that a level creates. In our data set, we add at most 1,846 intra-edges (when three abstract levels are defined) to an initial graph having 4,469 nodes and 16,420 edges. Assuming that a node and an edge occupy about the same amount of memory, we obtain an overhead of 8.83 percent.

The way that the overhead translates in terms of memory bytes is highly dependent on factors such as implementation, compiler optimizations, or size of the problem map. For instance, if the map size is at most 256×256, then storing the coordinates of a graph node takes two bytes. More memory is necessary for larger maps.

Since abstract nodes and edges are labeled by their level, the memory necessary to store an element might be larger in the abstract graph than in the initial graph. This additional requirement, called the *level overhead*, can be as little as 2 bits per element, corresponding to a largest possible number of levels of 4. Since most compilers round the bit-size of objects to a multiple of 8, the level overhead could actually not exist in practice.

The storage utilization can be optimized by keeping in memory (e.g., the cache) only those parts of the graph that are necessary for the current search. In the hierarchical framework, we need only the subgraph corresponding to the level and the area where the current search is performed.

For example, when the main abstract search is performed, we can drop the low-level problem graph, greatly reducing the memory requirements for this search.

The worst-case scenario for a cluster is when blocked tiles and free tiles alternate on the border, and any two border nodes can be connected to each other. Assume the size of the problem maze is $m \times m$, the maze is decomposed into $c \times c$ clusters, and the size of a cluster is $n \times n$. In the worst case, we obtain $4n/2 = 2n$ nodes per cluster. Since each pair of nodes defines an intra-edge, the number of intra-edges for a cluster is $2n(2n - 1)/2 = n(2n - 1)$. This analysis is true for clusters in the middle of the maze. We do not define abstract nodes on the maze edges, so marginal clusters have a smaller number of abstract nodes. For the cluster in a maze corner, the number of nodes is n and the number of intra-edges is $n(n - 1)/2$. For a cluster on a maze edge, the number of nodes is $1.5n$ and the number of intra-edges is $1.5n\ (1.5n - 1)/2$. There are four corner clusters, $4c - 8$ edge clusters, and $(c - 2)^2$ middle clusters. Therefore, the total number of abstract nodes is $2m(c - 1)$. The total number of intra-edges is $n(c - 2)^2(2n - 1) + 2(n - 1) + 3(c - 2)(1.5n - 1)$. The number of inter-edges is $m(c - 1)$.

Storage for the A* Open List

Since hierarchical path-finding decomposes a problem into a sum of small searches, the average size of open in A* is smaller in hierarchical search than in low-level search. Table 1.3 compares the average length of the open list in low-level search and hierarchical search. The average is performed over all searches described in the following section, without refining the results after the solution length. The data shows a three-fold reduction of the list size between the low-level search and the main search in the abstracted framework.

TABLE 1.3 Average size of the open list in A*. For hierarchical search, we show the average open size for the main search, the *SG* search (i.e., search for inserting *S* and *G* into the abstract graph), and the refinement search.

	Low level	Abstract		
		Main	*SG*	Refinement
Open Size	51.24	17.23	4.50	5.48

Experimental Results

Experimental Setup

Experiments were performed on a set of 120 maps extracted from BioWare's game *Baldur's Gate*, varying in size from 50×50 to 320×320. For each map, 100 searches were run using randomly generated S and G pairs where a valid path between the two locations existed.

The atomic map decomposition uses *octiles*. Octiles are tiles that define the adjacency relationship in four straight and four diagonal directions. The cost of vertical and horizontal transitions is 1. Diagonal transitions have the cost set to 1.42. We do not allow diagonal moves between two blocked tiles. Entrances with width less than 6 have one transition. For larger entrances, we generate two transitions.

The code was implemented using the University of Alberta Path-finding Code Library (*www.cs.ualberta.ca/~games/pathfind*). This library is used as a research tool for quickly implementing different search algorithms using different grid representations. Because of its generic nature, there is some overhead associated with using the library. All times reported in this paper should be viewed as generous upper bounds on a custom implementation.

The timings were performed on a 800 MHz Pentium III with 3 GB of memory. The programs were compiled using gcc version 2.96, and were run under Red Hat Linux version 7.2.

Analysis

Figure 1.5 compares low-level A* to abstract search on hierarchies with the maximal level set to 1, 2, and 3. The left graph shows the number of expanded nodes and the right graph shows the time. For hierarchical search we display the total effort, which includes inserting *S* and *G* into the graph, searching at the highest level, and refining the path. The real effort can be smaller since the cost of inserting *S* or *G* can be amortized for many searches, and path refinement is not always necessary. The graphs show that, when complete processing is necessary, the first abstraction level is sufficient for the map sizes that we used in this experiment. We assume that, for larger maps, the benefits of more levels would be more significant. The complexity reduction can become larger than the overhead for adding the level. As we show next, more levels are also useful when path refinement is not necessary and *S* or *G* can be used for several searches.

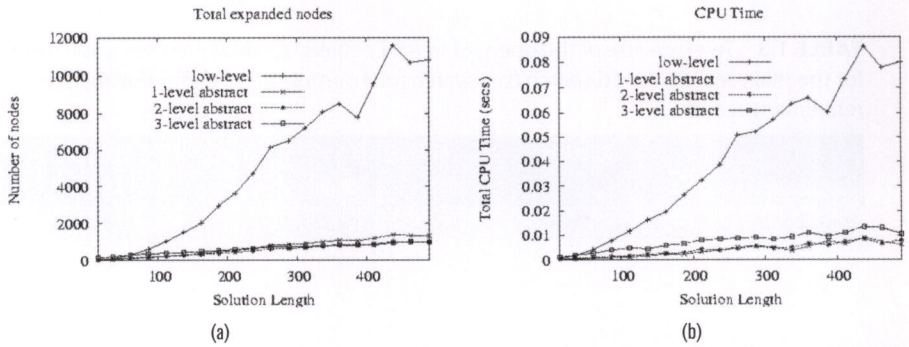


FIGURE 1.5 Low-level A* vs. hierarchical path-finding.

Even though the reported times are for a generic implementation, it is important to note that for any solution length the appropriate level of abstraction was able to provide answers in less than 10 milliseconds on average. Through length 400, the average time per search was less than 5 milliseconds on a 800 MHz machine.

A* is slightly better than HPA* when the solution length is very small. A small solution length usually indicates an easy search problem, which A* solves with reduced effort. The overhead of HPA* (e.g., for inserting *S* and *G*) is in such cases larger than the potential savings that the algorithm could achieve. A* is also better when *S* and *G* can be connected through a "straight" line on the grid. In this case, using the Euclidian distance as heuristic provides perfect information, and A* expands no nodes other than those that belong to the solution.

Figure 1.6 shows how the total effort for hierarchical search is composed of the abstract effort, the effort for inserting S and G, and the effort for solution refinement. The cost for finding an abstract path is the sum of only the main cost and the cost for inserting S and G. When S or G is reused for many searches, only part of this cost counts for the abstract cost of a problem. Considering these, Figure 1.6 shows that finding an abstract path becomes easier in hierarchies with more levels.

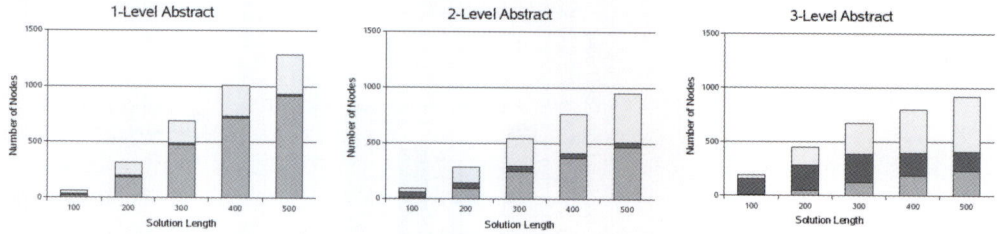


FIGURE 1.6 The effort for hierarchical search in hierarchies with one abstract level, two abstract levels, and three abstract levels. We show in what proportion the main effort, the *SG* effort, and the refinement effort contribute to the total effort. The gray part at the bottom of a data bar represents the main effort. The dark part in the middle is the *SG* effort. The white part at the top is the refinement effort.

Figure 1.7 shows the solution quality. We compare the solutions obtained with hierarchical path-finding to the optimal solutions computed by low-level A*. We plot the error before and after path-smoothing. The error measures the overhead in percents and is computed with the following formula:

$$e = \frac{hl - ol}{ol} \times 100$$

where hl is the length of the solution found with HPA*, and ol is the length of the optimal solution found with A*. The error is independent of the number of hierarchical levels. The only factor that generates suboptimality is not considering all the possible transitions for an entrance.

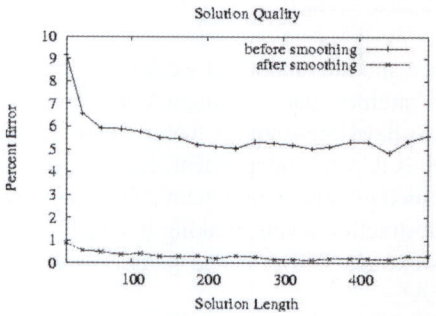


FIGURE 1.7 The solution quality.

The cluster size is a parameter that can be tuned. We ran our performance tests using 1-clusters with size 10×10. This choice at level 1 is supported by the data presented in Figure 1.8. This graph shows how the average number of expanded nodes for an abstract search changes with varying the cluster size. While the main search reduces with increasing cluster size, the cost for inserting S and G increases faster. The expanded node count reaches a minimum at cluster size 10.

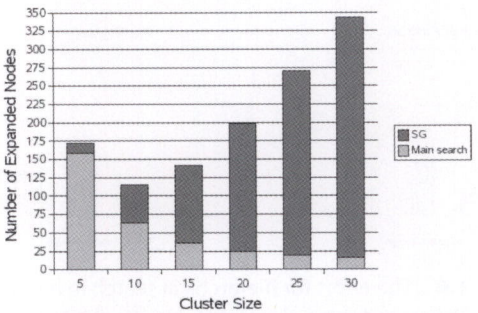


FIGURE 1.8 The search effort for finding an abstract solution. SG represents the cost of inserting *S* and *G*. The main search finds an abstract path in the abstract graph.

For higher levels, an l-cluster contains 2×2 $(l - 1)$-clusters. We used this small value since, when larger values are used, the cost for inserting S and G increases faster than the reduction of the main search. This tendency is especially true on relatively small maps, where smaller clusters achieve good performance and the increased costs for using larger clusters may not be amortized. The overhead of inserting S and G results from having to connect S and G to many nodes placed on the border of a large cluster. The longer the cluster border, the more nodes to connect to. We ran similar tests on randomly generated maps. The main conclusions were similar but, because of lack of space, we do not discuss the details in this article.

Conclusions and Future Work

Despite the importance and the amount of work done in path-finding, there are not many detailed publications about hierarchical path-finding in commercial games.

In this article, we have presented a hierarchical technique for efficient near-optimal path-finding. Our approach is domain independent, easy to apply, and works well for different kinds of map topologies. The method adapts to dynamically changing environments. The hierarchy can be extended to several abstraction levels, making it scalable for large problem spaces. We tested our program using maps extracted from a real game, obtaining near-optimal solutions significantly faster than low-level A*.

We have many ideas for future work in hierarchical path-finding. We plan to optimize the way that we insert S and G into the abstract graph. As Figure 1.6 shows, these costs increase significantly

with adding a new abstraction layer. One strategy for improving the performance is to connect S only to a sparse subset of the nodes on the border, maintaining the completeness of the abstract graph. For instance, if each "unconnected" node (i.e., a node on the border to which we did not try to connect S) is reachable in the abstract graph from a "connected" node (i.e., a node on the border to which we have connected S), then the completeness is preserved. Another idea is to consider for connection only border nodes that are on the direction of G. However, this last idea does not guarantee the completeness and it is difficult to evaluate the benefits beforehand. If the search fails because of the graph incompleteness, we have to perform it again with the subset of border nodes gradually enlarged.

The clustering method that we currently use is simple and produces good results. However, we also want to explore more sophisticated clustering methods. An application-independent strategy is to automatically minimize some of the clustering parameters such as number of abstract clusters, cluster interactions, and cluster complexity (e.g., the percentage of internal obstacles).

Acknowledgment

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE). We thank all members of the Path-finding Research Group at the University of Alberta. Markus Enzenberger and Yngvi Björnsson wrote a generic path-finding library that we used in our experiments. BioWare kindly gave us access to the *Baldur's Gate* maps.

Appendix

In this appendix, we provide low-level details about our hierarchical path-finding technique, including the main functions in pseudocode. The code can be found at the web site *www.cs.ualberta.ca/~adib/*. First, we address the pre-processing, and next the online search.

Pre-Processing

Figure 1.9 summarizes the pre-processing. The main method is *preprocessing*(), which abstracts the problem maze, builds a graph with one abstract level, and, if desired, adds more levels to the graph.

Abstracting the Maze and Building the Abstract Graph

At the initial stage, the maze abstraction consists of building the 1-clusters and the entrances between clusters. Later, when more levels are added to the hierarchy, the maze is further abstracted by computing clusters of superior levels. In the method *abstractMaze* (), $C[1]$ is the set of 1-clusters, and E is the set of all entrances defined for the map.

The method *buildGraph*() creates the abstract graph of the problem. First, it creates the nodes and the inter-edges, and next builds the intra-edges. The method *newNode*(e, c) creates a node contained in cluster c and placed at the middle of entrance e. For simplicity, we assume we have one transition per entrance, regardless of the entrance width. The methods *getCluster*1(e, l) and *getCluster*2(e, l) return the two adjacent l-clusters connected by entrance e. We use the methods *addNode*(n, l) to add node n to the graph and set the node level to l, and *addEdge*(n_1, n_2, l, w, t) to

```
void abstractMaze(void) {
  E = ∅;
  C[1] = buildClusters(1);
  for (each c1, c2 ∈ C[1]){
    if (adjacent(c1, c2))
      E = E U buildEntrances(c1, c2);
  }
}
void buildGraph(void) {
  for (each e ∈ E) {
    c1 = getCluster1(e, 1);
    c2 = getCluster2(e, 1);
    n1 = newNode(e, c1);
    n2 = newNode(e, c2);
    addNode(n1, 1);
    addNode(n2, 1);
    addEdge(n1,n2, 1, 1, INTER);
  }
  for (each c ∈ C[1]) {
    for (each n1, n2 ∈ N[c], n1 ≠ n2) {
      d = searchForDistance(n1,n2,c);
      if(d < ∞)
        addEdge(n1, n2, 1, d, INTRA);
    }
  }
}
```

```
void addLevelToGraph(int l) {
  C[l] = buildClusters(l);
  for (each c1, c2 ∈ C[l]) {
    if (adjacent(c1, c2) = = false)
      continue;
    for (each e ∈ getEntrances(c1, c2)) {
      setLevel(getNode1(e), l);
      setLevel(getNode2(e), l);
      setLevel(getEdge(e), l);
    }
  }
  for (each c ∈ C[l])
    for (each n1, n2 ∈ N[c], n1 ≠ n2) {
      d = searchForDistance(n1, n2, c);
      if (d < ∞)
        addEdge(n1, n2, l, d, INTRA)
    }
}

void preprocessing(int maxLevel) {
  abstractMaze();
  buildGraph();
  for (l = 2; l ≤ maxLevel; l++)
    addLevelToGraph(l);
}
```

FIGURE 1.9 The pre-processing in pseudo-code. This phase builds the multi-level graph, except for *S* and *G*.

add an edge between nodes n_1 and n_2. Parameter w is the weight, l is the level, and $t \in$ {INTER, INTRA} shows the type (i.e., inter-edge or intra-edge) of the edge.

The last part of the method *buildGraph*() adds the intra-edges. The method *searchForDistance*(n_1,n_2,c) searches for a path between two nodes and returns the path cost. This search is optimized as shown later in this article.

Creating Additional Graph Levels

The hierarchical levels of the multilevel abstract graph are built incrementally. Level 1 has been built at the previous phase. Assuming that the highest current level is $l-1$, we build level l by calling the method *addLevelToGraph* (l). We group clusters at level $l-1$ to form a cluster at level l (the method *buildClusters*(l), $l > 1$). C[l] is the set of l-clusters. The last part of the method *addLevelToGraph*() adds new intra-edges to the graph. Given a cluster c, $N[c]$ is the set of the nodes placed on the border of c that communicate to the outside of the cluster through inter-edges.

The Online Search

Finding an Abstract Solution

Figure 1.10 summarizes the steps of the online search.

```
void connectToBorder(node s, cluster c){
    l = getLevel(c);
    for(each n ∈ N[c])
        d = searchForDistance(s, n, c);
        if(d < ∞)
            addEdge(s, n, d, l, INTRA);
}
void insertNode(node s, int maxLevel){
    for(l = 1;l ≤ maxLevel; l++){
        c = determineCluster(s, l);
        connectToBorder(s, c)
    }
    setLevel(s, maxLevel);
}
path hierarchicalSearch(node s, g, int l){
    insertNode(s, l);
    insertNode(g, l);
    absPath = searchForPath(s, g, l);
    llPath = refinePath(absPath, l);
    smPath = smoothPath(llPath);
    return smPath;
}
```

FIGURE 1.10 The online processing in pseudocode.

The main method is *hierarchicalSearch*(*S*,*G*,*maxLevel*), which performs the online search. First we insert *S* and *G* into the abstract graph, using the method *insertNode*(*node*,*level*). Inside this, the method *connectToBorder*(*n*,*c*) adds edges between node *n* and the nodes that belong to *N*[*c*].

The method *searchForPath*(*S*,*G*,*maxLevel*) performs a search at the highest abstraction level to find an abstract path from *S* to *G*. If desired, we refine the path to a low-level representation using the method *refinePath*(*absPath*). Finally, the method *smoothPath*(*llPath*) improves the quality of the low-level solution.

Searching in the Multilevel Graph

In a multilevel graph, search can be performed at various abstraction levels. Searching at level *l* reduces the search effort by exploring only a small subset of the graph nodes. The higher the level, the smaller the part of the graph that can potentially be explored. When searching at a certain level

l, the rules that apply for node expansion are the following. First, we consider only nodes having levels greater than or equal to l. Second, we consider only intra-edges having level l and inter-edges having levels $\geq l$.

The search space can be further reduced by ignoring the nodes outside a given cluster. This is useful in situations such as connecting S or G to the border of their clusters, connecting two nodes placed on the border of the same cluster, or refining an abstract path.

REFERENCES

[Chen 1995] Chen, D. Z., R. J. Szczerba, and J. J. Urhan Jr. "Planning Conditional Shortest Paths Through an Unknown Environment: A Framed-Quadtree Approach." *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and System Human Interaction and Cooperation*, vol. 3 pp. 33–38, 1995.

[Holte 1996] Holte, R., M. Perez, R. Zimmer, and A. MacDonald, "Hierarchical A*: Searching Abstraction Hierarchies Efficiently." *Proceedings AAAI-96*, pp. 530–535, 1996.

[Korf 1985] Korf, R., "Depth-first Iterative Deepening: An Optimal Admissible Tree Search." *Artificial Intelligence*, 97: pp. 97–109, 1985.

[Moore 1999] Moore, A., L. Baird, and L. Kaelbling, "Multi-Value-Functions: Efficient Automatic Action Hierarchies for Multiple Goal MDPs." *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '99)*, pp. 1316–1321, 1999.

[Rabin 2000] Rabin, S., "A* Aesthetic Optimizations." In *Game Programming Gems*, ed. M. Deloura, pp. 264–271. Charles River Media, 2000.

[Rabin 2000] Rabin, S., "A* Speed Optimizations." In *Game Programming Gems*, ed. M. Deloura, pp. 272–287. Charles River Media, 2000.

[Reese] Reese, B. and B. Stout, "Finding a pathfinder". *http://citeseer.nj.nec.com/ reese99finding.html*.

[Samet 1988] Samet, H., An Overview of Quadtrees, Octrees, and Related Hierarchical Data Structures. NATO ASI Series, vol. F40, 1988.

[Shekhar 1997] Shekhar, S., A. Fetterer, and B. Goyal, "Materialization Trade-Offs in Hierarchical Shortest Path Algorithms." In *Symposium on Large Spatial Databases*, pp. 94–111, 1997.

[Stout 1996] Stout, B., "Smart Moves: Intelligent Pathfinding." *Game Developer Magazine*, October/November 1996.

[Tozour 2002] Tozour, P., "Building a Near-Optimal Navigation Mesh." In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 171–185. Charles River Media, Inc., 2002.

[Yahja 1998] Yahja, A., A. Stentz, S. Singh, and B. Brummit, "Framed-Quadtree Path Planning for Mobile Robots Operating in Sparse Environments."*Proceedings, IEEE Conference on Robotics and Automation, (ICRA)*, Leuven, Belgium, May 1998.

VERSATILE WALK ENGINE

Ronan Boulic, Branislav Ulicny, Daniel Thalmann
Virtual Reality Laboratory
I&C, EPFL, CH-1015 Lausanne, Switzerland
ronan.boulic@epfl.edu
branislav.ulicny@epfl.edu
Daniel.thalmann@epfl.edu

ABSTRACT

Walking is one of the most characteristic motions of humans, and the walking animation is indispensable for any game featuring humans or human-like characters. We propose an efficient walk component for the reactive animation of biped characters of any size and proportions. The movement can be controlled by changing independently its style, desired speed, and desired target position. Compared to traditional approaches, our gait style parameterization provides an efficient way to generate a multitude of varied walking animations, as needed, for example, by crowd simulations (as seen in Figure 2.1). One of our key specifications is to allow changes at any time as natural-looking movements tend to fluctuate over time. The resulting challenge we address is to maintain the smoothness of the gait and the coherence of the steps while the speed and style parameters are changing. Combined with the introduction of an angular speed parameter, we show how the walk component deals with speed and position control to ease higher-level behavioral control. Finally, we demonstrate its potential in game-like settings: a group of nonplayer characters is controlled by a rule-based behavioral module, and a player character is interactively controlled by a gamepad.

FIGURE 2.1 Walking crowd with different gaits.

INTRODUCTION

Human motion has always been difficult to simulate with acceptable level of physical realism and/or believability. It is especially the case in an interactive context, when high update rates are imposed. Today, only games with a narrow focus, like sport games, can manage the complexity of handling all the potential motion transitions a character offers at runtime. This is no more the case for games with a wider scope, like online games where we observe a significant gap between some high-quality character modeling and rendering on the one hand, and the corresponding lack of reactivity and fluidity of the animation on the other hand. The purpose of this article is to describe the structure of a generic walk animation component aiming at reproducing the coherence and the fluidity of the original motion while being able to react, on the fly, to changes in speed and goal.

Let us now expose the motivations and criteria grounding our design decisions before entering a more detailed description in the following sections. First, although motion capture is to date the best approach to obtain the fine dynamics of human movements, it is only the first stage in the process of producing exploitable animation data [Menache 2000]. The plain reuse of captured motions on a variety of characters is seldom possible too—the animation reuse is 100 percent effective only if two characters have the same number of joints, same segment proportions, and the same skinning. Otherwise, artifacts like self-collisions, collisions with the floor, floating in the air, or feet sliding are the most likely fate of this approach. For this reason, one major specification of a versatile walk engine is *genericity*, in other words, the ability to handle biped characters of any size and proportions. The section *Background* describes how temporal and spatial normalizations achieve this goal for the straight line case.

In a real-time *reactive* context, like games, the second most important criterion is the *performance*. We want to be able to change the desired speed or target position at any time, on the fly, without being penalized by additional computing cost. Our architecture is consistent with findings from neurophysiology supporting that walking can be seen as a motion pattern managed at the spinal cord level while the central brain is only in charge of modulating it according to the current context [Berthoz 1996]. This is very stimulating, as relatively simple methods exist for building motion patterns (see [Multon 1999] for a review). Associated with a dynamic time warping scheme, it is possible to synthesize a straight line walking animation with varying linear speed [Boulic 1990]. We extend this limited context to consider also a varying angular speed, a paramount variable when controlling a biped to reach a target position.

To summarize these premises, our walk animation component is designed to produce movement on the fly, without end, in a *continuously changing context* (e.g., where linear speed, angular speed, or target position change over time), and for a high number of differing biped characters.

While making critical design decisions to achieve this goal, we quickly faced a compromise: privileging the enforcement of the smoothness of the motion or the enforcement of the feet con-

straints. Our choice is based on the assumption that our sense of motion perception is highly sensitive to high frequencies; in other words, rapid variations [Bindiganavale 1998]. On the one hand, we can easily detect small glitches in the motion that immediately destroy its believability. On the other hand, we are less sensitive to the absolute altitude of the feet; there is a margin within which we are not perturbed by errors. Hence, it is more critical to ensure an overall *smooth* animation preserving the motion pattern quality rather than introducing continuity breaks due to feet constraint enforcement (this effect is better known as the *magnetic shoes syndrome*). Nevertheless, even if feet placement has a lower design priority in our architecture, we still handle it indirectly through the modulation of the motion pattern. This approach minimizes most of the discrepancies at a much lower computing cost than approaches based on Inverse Kinematics (IK) [Sun 2001]. A comparison with methods based on motion reuse is developed in the section *Discussion* later in the article.

The last key element for gait generation is the ability to design individualized gait style while remaining compliant with all the aforementioned criteria. Our approach remains simple while allowing for style variation depending on the context or the speed.

This article is structured as follows: first, a background section recalls the major elements for designing a straight line walk animation with dynamic time warping. Then, we briefly indicate how to map the generic angles produced by the walk component on the family of H-Anim compliant skeletons. The second half of the article describes how to go beyond the straight line walking model by studying the issues of handling the angular speed, passing through desired locations, and managing the gait style while minimizing feet discrepancies. We demonstrate the applications of our walk engine in game-like settings: a group of nonplayer characters is controlled by a rule-based behavioral module, and a player character is interactively controlled by a gamepad. Finally, we discuss how the proposed approach compares with the major references in locomotion animation and we outline our future research directions.

Background: Straight Line Walk Animation

Cycle Structure

Walking is a cyclic activity consisting in the repetition of a two-step motif as illustrated with its major key events on the top row of Figure 2.2. We adopted the convention from biomechanics that sets the start of the walking cycle on the `Left Heel Strike` event. The walking activity can be characterized as a *motion pattern* due to its stable structure composed of the alternation of supporting leg with an intermediate double support transition (Figure 2.2, middle row).

Temporal Normalization

The most convenient representation for our purpose is the *normalized cycle*, where all the walking cycle events are mapped on the interval [Berthoz 1996; Bindiganavale 1998]. The associated *phase* variable φ indicates the progression in the cycle. It is interesting to note that almost all the walking cycle events have a constant phase value whatever the speed (Figure 2.2, bottom line). This property motivated us to model the joint trajectories over such a normalized interval. A second justification for maintaining the phase variable is the high usefulness of events for high-level control, like setting when a character is allowed to accelerate or decelerate, turn right, or turn left.

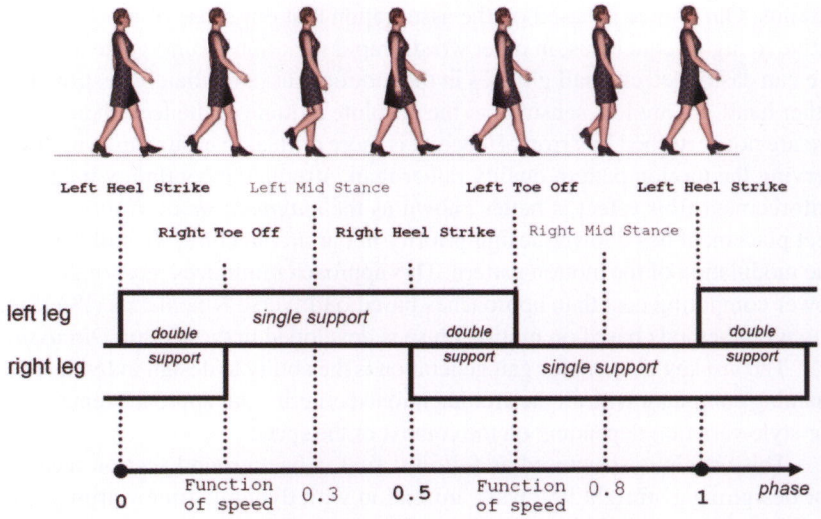


FIGURE 2.2 The walk cycle: key postures corresponding to the major walking cycle events (top row); structure of the walking activity in terms of an alternation of supporting phases (middle row); normalized cycle over [0, 1] with the key events' phase (bottom row). Only the `Toe_Off` event depends on the speed: `Right Toe Off` phase varies between 0.25 for null speed to 0.10 for fast walking speed.

Although time normalization is an important stage in the motion modeling, it is equally important to model how the *cycle duration* D_C evolves with speed because it is the key for producing a natural motion flow from the normalized joint trajectories representation. Inman has measured the cycle duration D_C of "normal" walking; in other words, the one a healthy adult naturally adopts [Inman 1981], and this for a wide range of *normalized speeds* V (V is obtained by dividing the speed by the leg length H defined as the distance between the hip joint center to the supporting surface in the standing posture). We use the following form of the resulting *Inman law* due to its numeric robustness for small values of V:

$$f = 0.743.\sqrt{V} \tag{1}$$

where f is the cycle frequency (the inverse of D_C). This equation is the key to the dynamic time warping scheme proposed in [Boulic 1990]. At runtime, whenever V changes, the cycle frequency f is updated with Equation 1. From this information and from the time increment Δt specified by the application level, the walk component computes the progression in the walking cycle through a phase increment $\Delta\varphi$ with:

$$\Delta\varphi = f.\Delta t \tag{2}$$

These two equations are sufficient to adjust dynamically the smooth and monotonic progression within the walking cycle while changing on the fly the current normalized speed V.

Ensuring Step Coherence

The spatial coherence of the walking pattern is ensured with the following equation relating V and f to the normalized cycle length L:

$$V = L.f \tag{3}$$

Figures 2.3 and 2.4 describe the key elements of our new step length approximation (half a cycle length) at a small fraction of the cost of computing the real 3D step length. Experience shows it provides an excellent coherence over the whole speed range. The approximation relies on the isosceles triangle formed at `Heel strike` by both legs; the angle at the hip side is given by the extrema of the hip flexion-extension amplitude. We approximate L as: $L = 2S/H$, which implies that:

$$L = 4\sin(\Delta_{hip} / 2) \tag{4}$$

An important aspect to notice is the shifted distribution of hip flexion maximum and minimum amplitudes with respect to the vertical; our choice of $hip_{max} = -2.hip_{min}$ is consistent with values observed in the natural walking pattern. It follows from Equations (1), (3), and (4) that the hip joint maximum amplitude Δhip is given by:

$$\Delta_{hip} = 2.\arcsin(0.335\sqrt{V}) \tag{5}$$

so the hip flexion-extension trajectory (Figure 2.5) is characterized with $hip_{max} = 2.\ \Delta hip/3$ and $hip_{min} = -\Delta hip/3$.

The section *Gait Styling* examines how to go beyond the rigid coupling between speed and step length defined by the Inman law; the resulting flexibility allows us to represent a wider

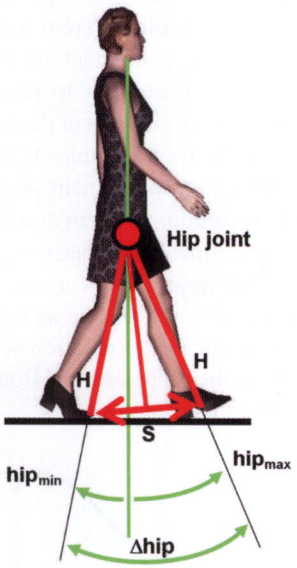


FIGURE 2.3 The Left Heel Strike event highlighting key elements for computing the normalized cycle length *L;* the approximation of doubling the normalized quantity *S/H,* (with *H* being the leg length), is easy to compute from the maximum hip flexion amplitude Δhip; the choice of hip_{max} being twice as big as hip_{min} reflects the natural walking pattern.

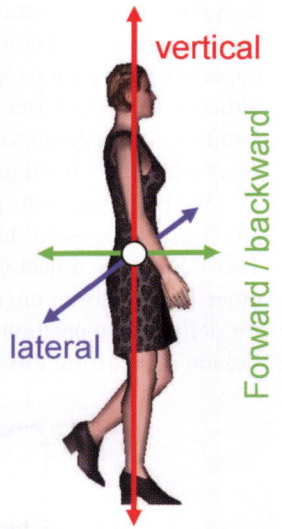


FIGURE 2.4 Directions of the three cyclic translations.

population of gait styles, from elderly persons who walk with higher step frequency at fast speed, to soldiers who have to produce a forced walk when marching in parade.

The Walking Pattern

At the joint trajectories level, the temporal and spatial normalizations yield similar benefits in genericity: the same joint trajectories are valid for all characters walking with the same values of V [Murray 1967]. As a consequence, it is sufficient to represent a joint trajectory over one normalized cycle [0, 1] and for a *small range of normalized speeds*; in our case, [0.0, 2.5] (unit is leg_length/second). Hence, each trajectory is a function of the phase Ṇ and the normalized speed V. Therefore, the straight line walking pattern is structured in:

A global displacement obtained by integrating the linear velocity over time.

A set of three cyclic translations expressed in the local frame of the character (see Figure 2.4), namely the *vertical*, *lateral*, and *forward/backward* translations. Although their amplitude is small, they are vital for the believability of the motion.

A body posture cycle (12 angles) expressed at body parts level like for the torso or the head, or directly as joint trajectories.

Most of the trajectories are flexion-extension trajectories with joint axes perpendicular to the plane of symmetry of the body (sagittal plane). Trajectories with different axes of rotation like lateral rotation (adduction-abduction) or internal rotation along a segment are necessary to model turning (see the section *Introducing an Angular Speed Parameter*) or to provide more natural upper body animation (see the section *The Spine Mapping*). Our movement decomposition, as one-dimensional trajectories, corresponds to data available in the biomechanics literature and is more intuitive for designing new gait styles. The amplitudes of the movement along each angle are small enough to avoid producing awkward posture when combining them at joint level.

We proposed in the previous section an efficient approximation for deriving the extrema of the hip flexion-extension. Figure 2.5 shows the corresponding trajectory over one normalized cycle. This is the only trajectory requiring such a precise update of the extrema as a function of V. The other 11 trajectories are also functions of V, but a simpler linear interpolation scheme is sufficient for defining the position of the extrema; see the details in the Appendix in [Boulic 1990] (the hip flexion model from this source should be superseded by the present one).

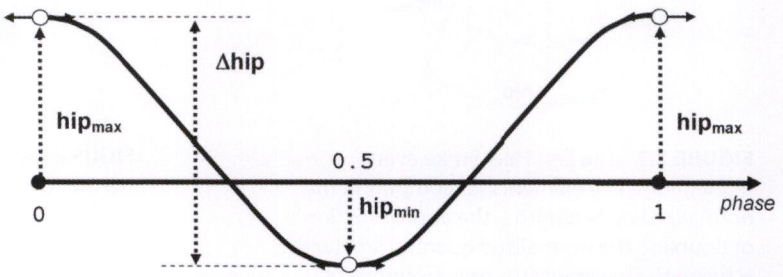


FIGURE 2.5 Normalized trajectory of the hip joint flexion-extension as a function of the minima identified in the step length approximation; cubic step functions are used between extremes.

Update of the Straight Line Walking Pattern

At runtime the user can specify the current speed V (in meter/s) while the environment generally defines the best time interval Δt guaranteeing a good frame rate; both information can change at each frame.

Figure 2.6 gathers the elements necessary for updating the generic posture and global location. As mentioned before, the heart of the model is normalized (stages included in the gray box), thus requiring normalizing the speed first. This explains the need for providing the character's leg length right from the start; in case no value is given, the walk component uses a default value of 1 meter (corresponding to the leg length of a man 1.88 m – 6.16 ft. tall).

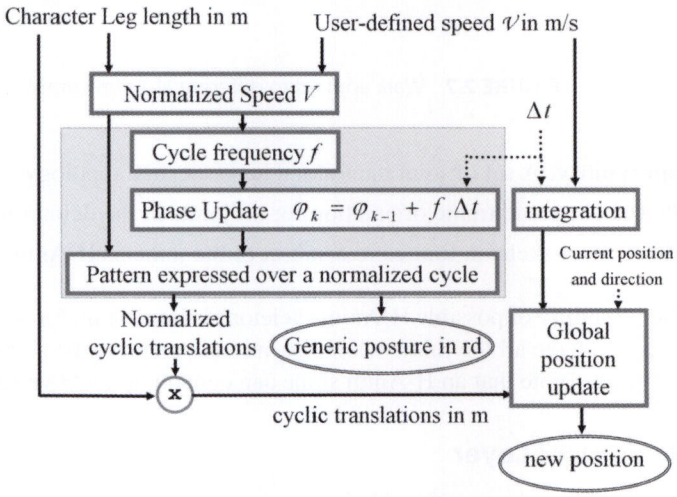


FIGURE 2.6 Runtime update of the straight line walking model.

The current speed is considered as an average speed; its integration over the time interval Δt provides the global displacement in the frontal direction (Figure 2.4). The walk engine updates first its instantaneous frequency and its phase prior to evaluating the joint values for the new normalized speed and phase values. Cyclic body-centric translations are multiplied by the character leg length prior to updating the global position. The mapping of the generic posture angles on the set of available character joints is briefly described in the next section.

Posture Mapping on H-Anim Compliant Skeletons

Interest of H-Anim

The H-Anim standard proposal aims at fostering motion reuse through common conventions for human-like creatures (see Figure 2.7) [H-Anim 2001]. Among its key aspects are:

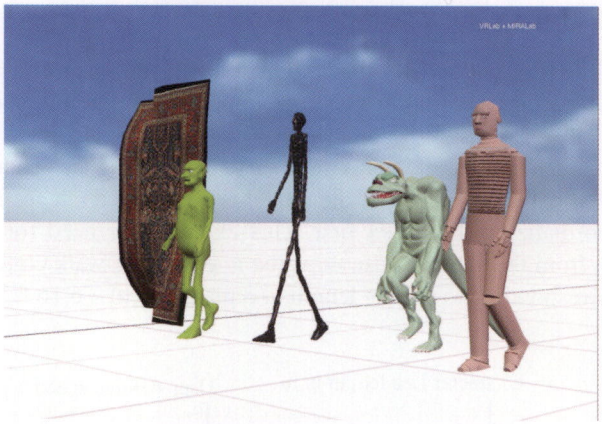


FIGURE 2.7 Walk applied to different H-Anim compliant characters.

Interoperability: A set of joint names and an associated topology.

Simplicity: The same frame orientation for all joints in the default posture.

Versatility: Any skeleton limited to a subset of the joints is H-Anim compliant too.

The versatility of possible H-Anim skeletons is interesting for scenarios where first rank characters need to have all the joints while second rank ones can be simplified to just the hip and the shoulder joints (note that an H-Anim spine can contain up to 23 vertebrae).

The Character Control Layer

The mapping of the walk posture on the character skeleton is handled in the *Character control layer* because we want to keep the walk component independent from the choice of character model (see Figure 2.8). At initialization stage, the *Character control layer* starts by establishing the list of existing joints that are animated by the walk animation; only these joint trajectories will be computed at runtime. This layer also computes the leg length by accumulating the following segment lengths (if they are defined): thigh, shank, ankle_to_subtalar. The resulting length is scaled by a factor taking into account the remaining distance between the lowest joint and the supporting surface (we use 1.1 for a human). The application level can always set a different value if necessary.

The Spine Mapping

At initialization time, we first count the number of vertebrae per spine region because each vertebra joint has a different mobility; Table 2.1 approximates their mobility potential along the three rotation axis tilt, roll, and torsion [White 1990]. The image in Table 2.1 left column shows the anatomic regions and the axis orientations. For example, we can see that the lumbar vertebrae, just above the pelvis, cannot move along the torsion axis while they can move uniformly for the roll axis and they have a decreasing mobility for the tilt axis from vl5 to vl1.

During online animation, the walk component provides three angles for the `torso_to_pelvis` and the `head_to_torso` orientations. These scalar values are distributed on the available vertebrae

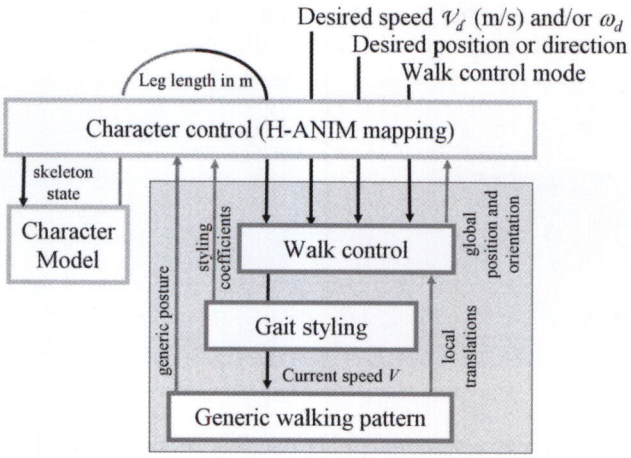


FIGURE 2.8 The walk action consists of the Character control layer driving the Walk Animation Component (gray box). The walk control (see the section *The Walk Control Layer*) is responsible for achieving smoothly high-level goals (desired speed, desired position, or direction). The gait styling (see the section *Gait Styling*) stores scaling and offset coefficients and maintains the step length coherence.

according to the distribution type specified in Table 2.1. The distribution coefficient formulas are gathered in Table 2.2. Once each individual vertebra joint has its proportions of the three orientations, they are combined into a quaternion for updating the model skeleton.

TABLE 2.1 Anatomic model of spine showing the type of mobility distribution for the three spine regions (from bottom to top: lumbar, thoracic, and cervical).

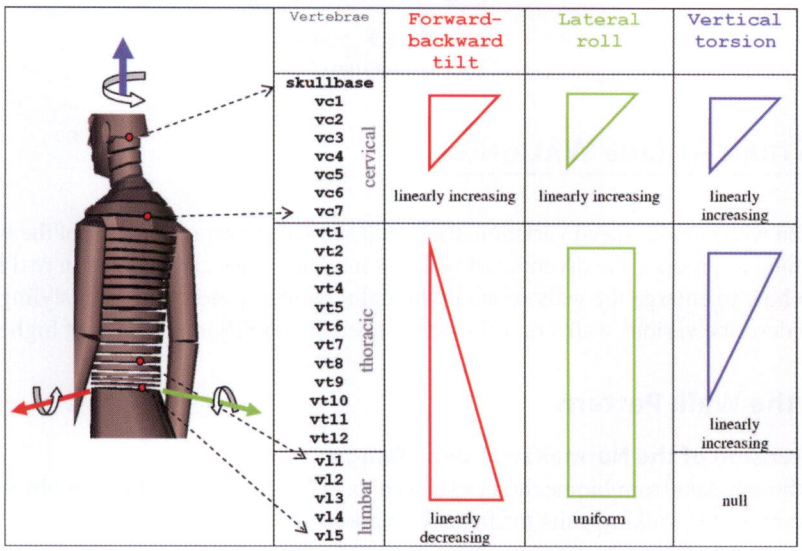


TABLE 2.2 Formulas giving the angle proportion for joint *i* (among a group of *n*) according to a type of mobility distribution; the index *i* starts with *i* = *0* on the pelvis side.

Uniform distribution	$C_i = \frac{1}{n}$
Linearly increasing	$C_i = i.(\frac{2}{n.(n+1)})$
Linearly decreasing	$C_i = (n+1-i).(\frac{2}{n.(n+1)})$

Figure 2.9 shows a spine mapping distribution on the full H-Anim spine; associated to the local body-centric translations and the gait styling, the resulting pelvis-to-head animation adds a lot to the character believability at a very low cost.

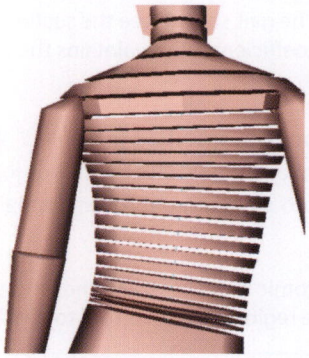


FIGURE 2.9 Spine mapping on the "salami man."

BEYOND STRAIGHT LINE WALKING

Even with smooth speed variation, the "adult statistical average" nature of the walking pattern is too limited to produce the diversity of walking movement we can observe in real life. Here, we examine how to enlarge the walk space in three directions: speed range, gait styling, and turning. Later, we describe various walk control modes aimed at simplifying the task of higher behavioral levels.

Enlarging the Walk Pattern

Extension of the Normalized Speed Range

Although data from biomechanics seldom exist for fast speeds, there is obviously a need for producing faster walking gaits for fantasy characters.

To offer this possibility, we have made the design choice of breaking the Inman law above the maximum normalized speed of V_{max} = 2.5 (leg_length/s) to replace it by a linear relationship between step frequency and normalized speed (see Figure 2.10). In other words, above V_{max} the step length remains constant while only the frequency augments to achieve the current speed. Our new maximum speed V_{maxf} corresponds to a maximum frequency of two cycles/s (i.e., four steps per second).

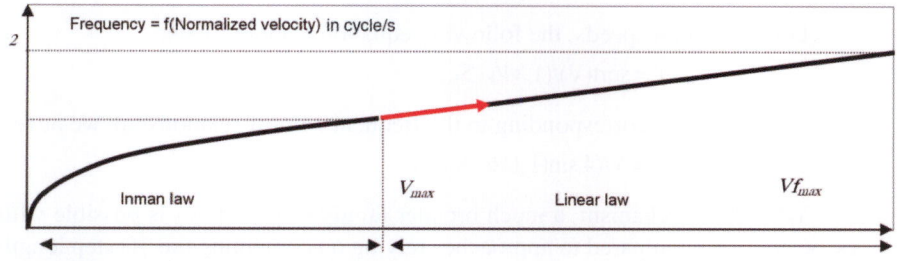


FIGURE 2.10 Extending the Inman law.

Gait Styling

The default walking pattern can be enriched by allowing to scale and offset the three body-centric translations and the 12 angles defining the generic posture (left and right sides can be characterized independently). This straightforward solution is appreciated by animators as long as they can immediately see the influence of their parameter tweaking on the gait style for the whole range of speeds. This is the purpose of the gait editor, whose interface is shown in Figure 2.11. The whole context can be adapted to users' needs (animated character, scene, linear and/or angular speed, treadmill option), and the set of parameters can be saved to files. At runtime, multiple gait styles can be preloaded and interpolated according to users' commands.

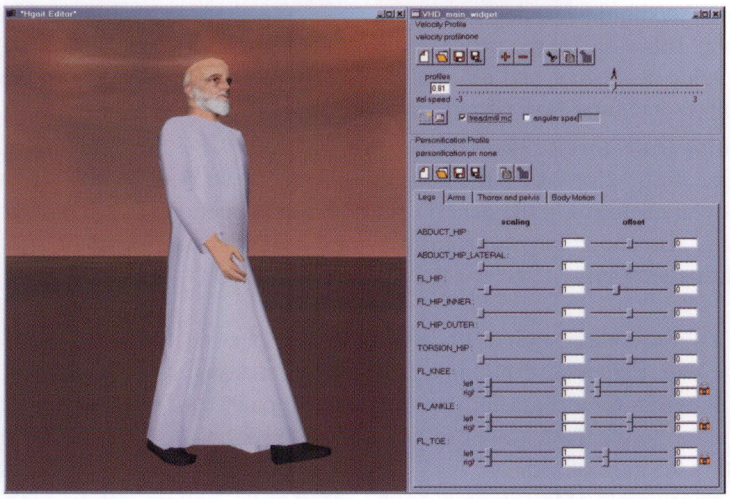


FIGURE 2.11 Gait editor interface.

The *hip flexion-extension scaling* style parameter, noted S_{hip}, is the key to change the default coupling between speed and step length; for a given normalized velocity V, it allows to make bigger or smaller steps (Figures 2.3 and 2.5). However, it requires a frequency adjustment to avoid the feet sliding artifact. The adjusted frequency f has to ensure that V is still achieved when the "normal" hip flexion amplitude Δhip is scaled by S_{hip}. Let us note L_S the cycle length corresponding to the scaled hip amplitude; according to equation 3 we have : $f = V / L_S$, so:

$$f = V/(4.\sin (S_{hip} . \Delta hip/2)) \quad (6)$$

For very small speeds, the following equation is more robust:

$$f = \text{sqrt}(V)/(1.346 . S_{hip}) \quad (7)$$

and for speeds corresponding to the frequency linear relationship, we have:

$$f = V/(4.\sin(1.116 . S_{hip})) \quad (8)$$

With this mechanism, a much broader family of gait styles is possible while minimizing the associated cost compared to approaches relying on computing the 3D step length (the section *Performances* provides some measurements quantifying the walk engine performances).

Introducing an Angular Speed Parameter

Addressing the production of believable curved walking animation is essential, because mapping a straight line walking pattern on a curved path leads to noticeable side shifts of the feet. An integrated approach is necessary, as we want to extend the speed control continuum, available in the straight line case, to curved trajectories. The extended model must also remain truly reactive, meaning that the turning characteristics can change at any time and instantaneously influence the walking pattern. We have retained the *angular speed* parameter ω to modulate the walk at three levels:

Walk control level (Figure 2.8): The angular speed ω is given precedence over the linear speed V to prevent excessive centrifugal acceleration (details in the section *The Walk Control Layer*).

Cycle frequency: Our working hypothesis states that a pure turning cycle must be completed within a maximum 180 degree turning angle; this leads to a minimal turning cycle frequency: $f_t = \omega/\pi$. Therefore, at the phase update stage (Equation 2 and Figure 2.6) we select the maximum value between f_t and the straight line frequency computed from Equation 6. This approach guarantees a natural flow of motion even for very slow linear speeds.

Hip joint trajectories: The angular speed influences the hip joint angles by modulating differently the hip flexion-extension depending on the inner/outer side of the leg, and by introducing two new trajectories in the generic pattern: lateral opening, called *adduction-abduction*, and twisting, also called *internal rotation*. We develop the evaluation of their extrema in the remainder of this section.

First, the combination of linear and angular speeds defines a *normalized radius of curvature R* as $R = V/\omega$. Figure 2.12 highlights the spatial and angular relationships at two successive `Heel Strike` events for a character moving on the circular path with constant radius R. On each side of the middle circular path is another arc—with radii R_{in}, R_{out}—separated by the same distance D_h representing the normalized half_hip_joints distance (Anthropometric tables give $D_h = 0.1$).

The principle of the mapping is described in the bottom-left drawing displaying a `Left Heel Strike` event: between the `Heel Strike` event and the `Left Mid Stance` event (Figure 2.2), the

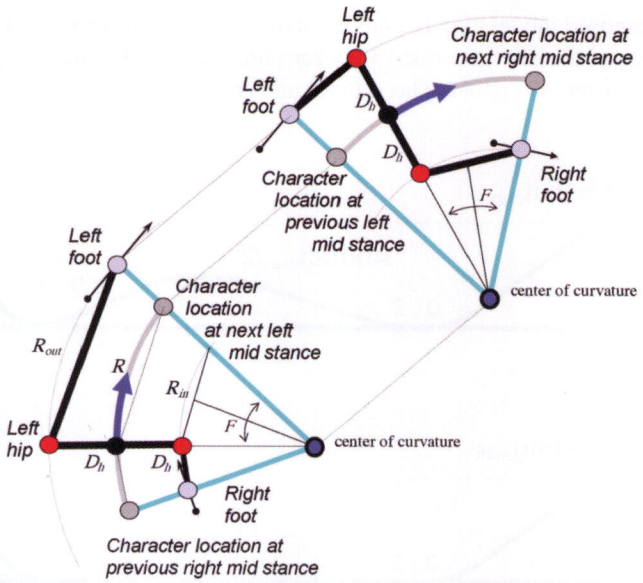


FIGURE 2.12 Bird's eye view of a `Left Heel Strike` event (bottom left) and the immediately following Right Heel Strike event (top right) for a clockwise turning speed (D_h is exaggerated for clarity).

central body point (black dot) moves over two-third of a step, like for the straight line walk (Figure 2.3) except the movement occurs now on the middle arc (the new location is the striped dot). Knowing that, at `Mid Stance`, the body is aligned with the supporting leg, we can approximate the location of the supporting foot at `Heel Strike` by drawing the radius going from the center of the curvature, through the `Mid Stance` location until it intersects the outer circle. We obtain the Left foot location indicated in the bottom drawing in Figure 2.12.

The same principle applies to determine the location of the right foot. By construction, the angle sector containing the front leg is twice the one containing the back leg. From this geometric construction, we can deduce the information we require. For a given linear speed V and style S_{hip}, the normalized body displacement from `Heel Strike` to `Mid Stance` is approximately: $D = sin(S_{hip}.hip_{max})$, as seen in Figure 2.3. Therefore, the front angular sector is given by $F = D/R$. Table 2.3 gathers the expressions of the angle maxima while Figure 2.13 gives the shape of normalized cycles for the two new angles (lateral and twisting).

TABLE 2.3 Value of the turning case maxima as functions of hip_{max}, the minima are half the value of the maxima for the lateral angle, together with opposite sign for the twisting.

Maximum Amplitude	Flexion-Extension	Lateral Angle Abduction-Adduction	Twisting Internal Rotation
Inner side	$asin((R - H_d)sinF)$	$asin(2(R - H_d)sin(F/2)^2)$	F with limitation
Outer side	$asin((R + H_d)sinF)$	$asin(2(R + H_d)sin(F/2)^2)$	F with limitation

The proposed model only intends to cover cases with positive R_{in} values (Figure 2.12); in particular, it is insufficient for turning with zero linear speed. Figure 2.13 gives the shape of normalized cycles for the two new angles (lateral and twisting).

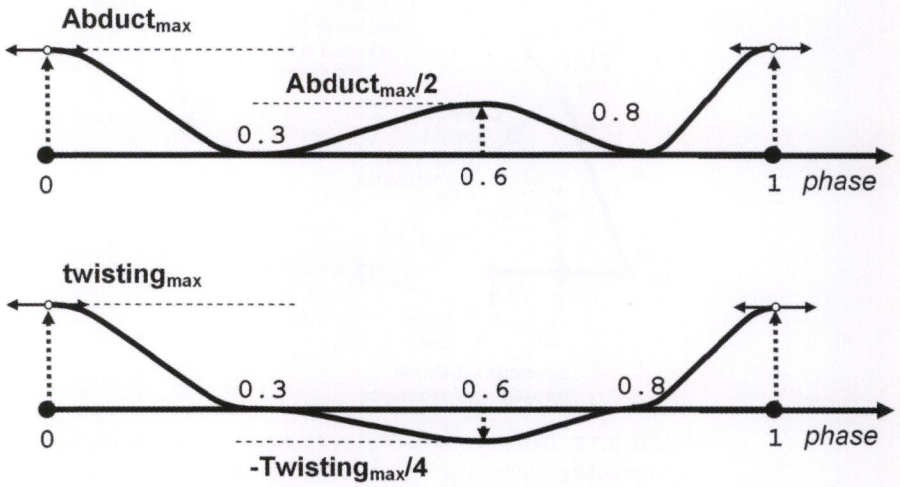


FIGURE 2.13 Normalized trajectories for the turning case adduction-abduction (top) and the twisting (bottom). Note: the twisting is generally offset by a little constant opening of the feet.

The Walk Control Layer

The main purpose of the walk control layer is to filter high-level discontinuities arising from sudden changes in the character goals. Indeed, it is perfectly valid for a player or an AI module to change at will the value of the desired linear and angular speeds. These changes can be directly executed with the previously seen architecture, or they can go through the walk control layer as presented now. This layer proposes two main control modes: speed and position. In both modes, the user has to specify a *desired linear speed* V_d that the control layer tries to achieve with a classical proportional-derivative controller where an acceleration term is derived from the difference between the desired and the current quantity. An aspect specific to the walking pattern is that some intervals in the cycle are not valid for accelerating or decelerating; hence, the scaling down factor applied to the acceleration shown in Figure 2.14. In the speed mode, the user can also specify a desired angular speed ω_d, also achieved with a proportional-derivative controller.

A mechanism for managing unrealistic combinations of linear and angular speed is necessary to produce believable gaits. We have chosen to give a higher precedence to angular speed achievement whenever its combination with the current linear speed produces a centrifugal acceleration greater than a normalized threshold a_c. This decision is motivated by the great impor-

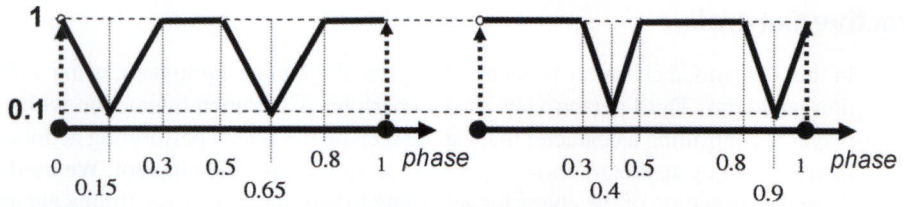


FIGURE 2.14 Scaling down the linear acceleration (left) or deceleration (right) as a function of φ.

tance of the angular speed to steer the character toward a goal position. Therefore, at any time we ensure that V_d is smaller than a temporary maximum desired linear speed V_{d_temp} given by:

$$V_{d\_temp} = a_c / \omega \tag{9}$$

In the position control mode, the angular speed is under the control of a proportional-derivative controller driven by the angular error between the character front direction and the direction toward the goal, similar to [Reynolds 1999]. There is one problem with constant values of the proportional factor and the damping terms: the character may miss the goal and have to U-turn and come back. Raising the proportional factor results in stiff turning behaviors. Our proposal is to make a simple estimation of whether the goal will be reached or not at each update. The prediction consists in prolonging the instantaneous arc of circle introduced in the section *Introducing an Angular Speed Parameter* and checking whether it intersects the tolerance circle around the target position. In case our prediction shows a miss, the proportional factor is slightly raised and this continues as long as the prediction is not successful. After reaching the target position, the proportional gain goes back to its normal value over a few iterations.

The walk control layer manages an internal update loop running at 25 Hz whenever the update interval Δt is greater than 0.04s. This is necessary to guarantee correct pursuit behavior when following dynamic targets. In addition, the position control mode offers two options regarding the behavior at the goal: either the walking character *goes through* the desired target (within the tolerance) and continues in a straight line with the user-defined desired speed, or the *halting* option is activated, which results in a specific coordination of speed and phase that is beyond the scope of this article.

RESULTS

We created two game-like scenarios to test the feasibility of our approach in conditions close to its intended use. The scenarios were built with VHD++, a real-time development framework for VR/AR systems featuring advanced virtual character simulation technologies [Ponder 2003]. The walk animation component is integrated into our humanoid animation component [Emering 2000].

All the results presented here (and Figures 2.1 and 2.7) have corresponding animation files on the article's Web page *at www.jogd.com*. The animation files were recorded from real-time sessions on a Pentium 4, 2.2 GHz machine with Quadro4 graphics card for all demos for this section. The performance tests were run with the resolution of 1600 × 1200 × 32. Animations for Figures 2.9 and 2.11 were made with slower machines.

Interactive Control

In this scenario, a character is controlled by standard game equipment, using a third-person interaction paradigm. Third person view of the character is common interaction with computer games: a player is controlling a character that can be seen on the screen performing actions according to commands given by keyboard, mouse, joystick, or any combination thereof. We used the walk engine to drive the animation of the character according to user inputs coming from a gamepad.

The user is directly controlling displacement and actions of the character in the virtual environment. The main challenge is to create, on the fly, coherent and believable walking animation while continuously taking into account the user inputs. Obviously, in such a dynamic environment, the animation cannot be precomputed.

A gamepad allows you to enter the input in several ways: it includes analog and digital joysticks, buttons, and a slider. For controlling the walking, we selected one analog two-axis joystick, leaving the other channels, for example, for triggering the actions or controlling of the camera.

We interpret the inputs from the joystick's axis according to two control metaphors—a character can either move relative to a fixed camera, or a camera follows a character. In the former case, the walk is directed by continuously setting proximate goals to reach using a position control mode (see the section *The Walk Control Layer*); in the latter case, we directly control frontal and angular speed of the walking in a speed control mode (see Figure 2.15).

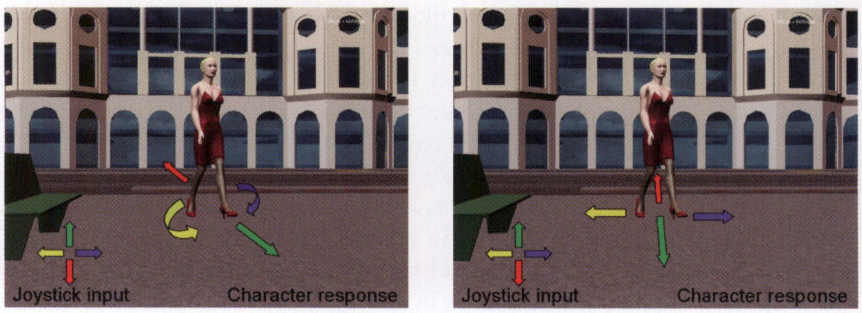
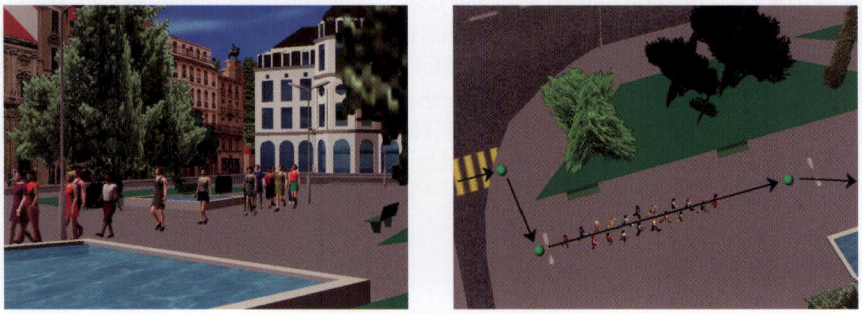


FIGURE 2.15 Gamepad control metaphors: a) speed, b) position.

The velocity of the walk is proportional to the analog values from the joystick's axis. In addition to the displacement control by joystick, we associated buttons with different gait styles, so that pressing the button triggers interactively a transition to a new walking style. The walk engine was able to handle continuous changes of the speed, direction, and gait style initiated from the user input while maintaining smoothness of the gait and coherence of the steps. This scenario was implemented using the DirectInput library [DirectX 2003] for reading values from the gamepad hardware and mapping them to actions.

Nonplayer Character Control

In the second scenario, a group of nonplayer characters moves semi-autonomously in the virtual environment (Figure 2.16). This scenario demonstrates interfacing a higher-level AI behavior with



(a) (b)

FIGURE 2.16 Group of characters controlled by behavior rules: a) side view, b) top view (showing path).

a low-level motion control provided by the walk component. Every character is controlled by a behavior engine that chooses behavior appropriate to the state of the agent and the state of the environment. In this article, we focus on the motion part of the model; the behavioral rules part is described in [Ulicny 2002]. The goal is to alleviate AI from micro-managing walk trajectories and animations. An interface between higher-level control and walk component is indeed very thin, consisting of only eight methods for controlling speed, goals, and gait.

The behavior computation is organized into several layers as shown in Figure 2.17. After higher levels decide that the next action should be displacement of the agent, a path-finding layer computes a path from the current to the target location avoiding static obstacles in the environment (e.g., buildings, trees, or benches) [Kallmann 2003]. The walk engine is then fed with a sequence of intermediate goals to reach using a position control mode (see the section *The Walk Control Layer*).

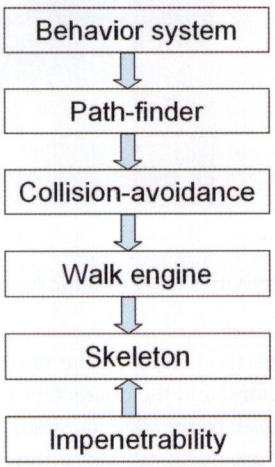


FIGURE 2.17 Motion control layers.

The collision avoidance layer tries to alter path-segments and walking speed in order to avoid potential collisions. However, as this is not always possible (and also would not be realistic—in the real world, sometimes we do collide), there is the physical impenetrability layer working directly with the skeleton position ensuring that the agents will not be in the same location at the same time, using a simple force-based model [Helbing 1995]. In the current implementation, a physical impenetrability layer can induce sliding of the feet in the case of the collision as it is working independently of the walk engine.

Performances

We performed several tests with different numbers of virtual humans and different virtual environment settings to assess the performance of the walk engine in the context of full application (Figure 2.18). We compared the overall performance of the application (refresh rate in frames per second) using a walk engine in position and speed control modes with baseline scenarios of a simple rendering of static characters and characters playing prerecorded keyframe animation sequence.

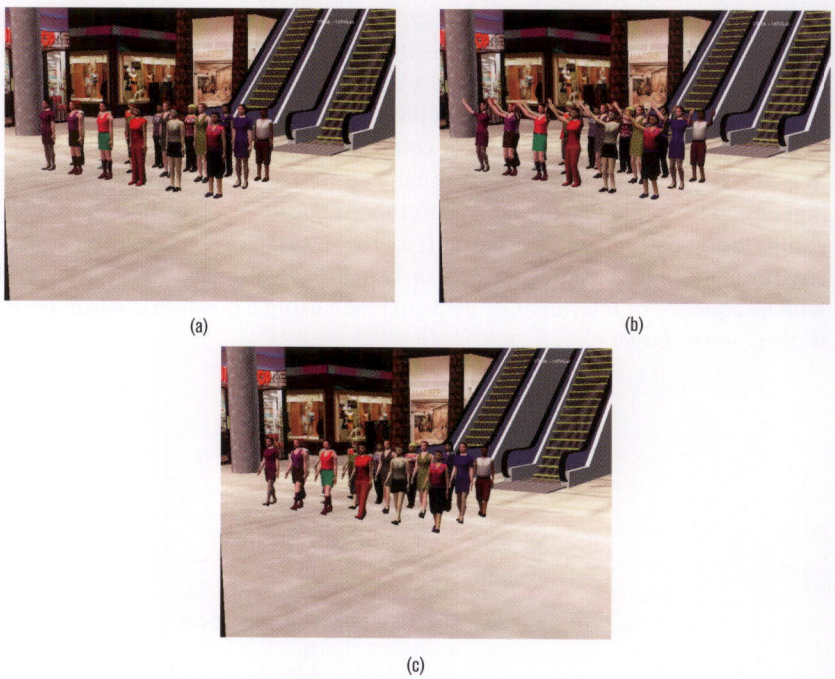


FIGURE 2.18 Performance tests: a) static humans, b) keyframe, c) walk.

Table 2.4 summarizes the results of the performance tests. All tests were performed in two settings—only the characters, and the characters plus the shopping mall scene. The influence of a walk engine on the overall frame rate is small and comparable to the influence of a simple keyframe playback; there are no significant differences between speed and position control modes of the walk. The factor with the biggest impact on performance was the complexity of the scene

(polygon count and texture sizes). After filtering all the costs due to the context (character skinning and scene rendering), the remaining computing time assignable to the sole walk engine shows that it can work at roughly 2 KHz.

TABLE 2.4 Performance test results (in frames per second). Most of the computing cost is due to the display of the characters and the scene (mall model adds 16.7K polygons to the polygon counts for humans shown in the table).

Num. of Humans	1 (2.2K polygons)		9 (17.2K polygons)		18 (30.5K polygons)	
Scenario	Empty	Mall	Empty	Mall	Empty	Mall
Still	180	17.9	34	12.3	20.5	9.9
Keyframe	160	17.9	29	11.6	17.3	9.2
Walk (speed)	165	17.8	29	11.5	17.5	9.4
Walk (position)	165	17.9	29	11.5	17.3	9.2

In a separate test, we evaluated the performance of the crowd engine and collision detection. The crowd behavior engine without collision avoidance can work at around 68 KHz per character, or 3.8 KHz for 18 characters (the complexity is linear in respect to the number of characters). Collision avoidance adds significantly to CPU costs, as its complexity is quadratic with respect to the number of characters. A crowd engine integrated with collision detection runs at around 2.2 KHz for 18 characters without optimization, or at approximately 3.2 KHz with a bin-lattice locality queries optimization [Reynolds 1999].

Memory requirements are also an important issue, as gaming applications can often use only limited resources. The walk component uses data structures with size of a few hundred bytes per character, compared to around 2,000 bytes for every skeleton keyframe of motion-captured animation. The memory used for walk synthesis is constant with respect to the length of the animation.

Discussion

The literature on walking animation is so rich that a full article would be necessary to discuss the advances since the last major review of the field [Multon 1999]. For this reason, we focus our comparison on the essential criteria a walk component should address to meet the stringent constraints of a game environment: ease of gait design, transparent biped-genericity, high online performance and reactivity, coherent motion believability, and variability.

The first two criteria are especially difficult to address for physically based approaches due to scaling issues when targeting a wide range of character sizes and morphotypes [Hodgins 1997], and to subtle control parameter tweaking [Laszlo 1996, 2000]. Simplified footprint-driven approaches have a great potential for computing displacements in complex environments [Van de Panne 1997; Choi 2000], but their higher computing cost may hinder their generalization.

Recently, significant advances have been made in the processing and the integration of captured motions within dedicated graph structures [Kovar 2002, 2003; Lee 2002]. These methods restitute the natural motion dynamics recorded in sets of motion clips and ensure smooth transitions between a subset of clips. An approach giving a higher control to the author for designing the graph

is described in [Gleicher 2003]. A high-performance specialized locomotion graph, integrating standing, running, and walking, is described in [Park 2002]; it relies on an efficient methodology for online retargeting developed in [Shin 2001]. The limitation of approaches based on captured motion comes from the dependency on a motion clip database. The required memory size might be a problem for some platforms, as the motion clip database has to sample, with sufficient density, the locomotion parameter space in speed, turning angle, style, and so forth. In addition, producing new stylistic variations seems difficult to generate on the fly; it requires an offline processing stage.

In comparison, we have presented various results highlighting the versatility of the walk engine in terms of the variety of the animated characters (Figures 2.7 and 2.9), smoothness of the motion with varying context, interactive control, and performances (Figures 2.1 and 2.18). An important aspect that could not be addressed completely is the management of natural variations over time, which is especially important for believable crowds. Even if a model produces perfect walking motion for a single character, it looks artificial if the same motion is applied to all the characters [Ulicny 2002]. We can already introduce some smooth randomness at the desired speed level, which is in turn distributed on all the low-level parameters. However, it will be worth adding motion texture to make the balancing limbs more lively [Pullen 2002]. At the gait style level, the walk engine provides a mechanism to create even more variety, but more work is necessary to create a large number of believable gaits for a crowd simulation automatically.

Conclusion and Future Work

Future work will concentrate on introducing lateral speed and local slope parameters in the same way as the current linear "frontal" speed and the angular speed. An open issue at the walk control level concerns the design of a fully reactive solution for "position and direction" goal. This is not as easy as it may seem, as we want to deal with dynamic goals and to handle properly the cases where classic algorithms based on cubic curves produce a trajectory cusp. In the longer term, we will explore online prediction of a few future steps for improving the adaptive control and planning problems like halting.

Acknowledgments

Research has been partly sponsored by the Swiss National Research Foundation and the Federal Office for Science and Education in the framework of the ERATO project for the control of groups. The monster model belongs to 3ds max Tutorial material. We want to thank Mireille Clavien for the other characters and environment design, Michal Ponder for the VHD++ test application and support, Sebastien Schertenleib for Python support, Sebastian Illan for gamepad control, Marcelo Kallmann for the path-finder, and skin deformation service provided by Frederic Cordier from MIRAlab.

REFERENCES

[Berthoz 1996] Berthoz A. "*Brain's sense of motion,*" Harvard University Press, 2002, translated from "*Le sense du movement*," Audile Jacob, 1996.

[Bindiganavale 1998] Bindiganavale R. and N. Badler, "Motion Abstraction and Mapping with Spatial Constraints," *Proceedings of Captech'98*, LNAI 1537, pp. 70–82, Springer-Verlag Berlin Heidelberg.

[Boulic 1990] Boulic R, D. Thalmann, and N. Magnenat-Thalmann, "A global human walking model with real time kinematic personification." *The Visual Computer,* vol. 6 (6), December 1990.

[Choi 2000] Choi, K-J. and H. -S. Ko, "On-line Motion Retargeting," *The Journal of Visualization and Computer Animation*, 11(5):223–235, 2000.

[DirectX 2003] DirectX homepage, *www.microsoft.com/directx*.

[Emering 2000] Emering L., R. Boulic, T. Molet, and D. Thalmann, "Versatile Tuning of Humanoid Agent Activity," *Computer Graphics Forum*, vol.19 (4), pp. 231–242, 2000.

[Gleicher 2003] Gleicher, M., H. J. Shin, L. Kovar, and A. Jepsen, "Snap-Together Motion: Assembling Run-Time Animation," *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pp. 181–188, ISBN:1-58113-645-5, Monterey, CA, 2003.

[H-Anim] Humanoid Animation Working Group, "The H-Anim 2001 specification," *www.hanim.org*.

[Helbing 1995] Helbing, D. and P. Molnar, "Social force model for pedestrian dynamics," *Phys. Rev.* E 51, 4282–4286 (1995).

[Hodgins 1997] Hodgins, J. K. and N. S. Pollard, "Adapting Simulated Behaviors for New Characters," *Proceedings of SIGGRAPH 97*, pp. 153–162, 1997.

[Inman 1981] Inman, V. T., H. J. Ralston, and F. Todd, *Human Walking*, Baltimore, Williams & Wilkins, 1981.

[Kallmann 2003] Kallmann, M., H. Bieri, and D. Thalmann, "Fully Dynamic Constrained Delaunay Triangulations," *Proceedings* of *Geometric Modelling for Scientific Visualization*, ISBN 3-540-40116-4, Springer-Verlag, Heidelberg, Germany.

[Kovar 2002] Kovar, L., M. Gleicher, and F. Pighin, "Motion Graphs," *ACM Transactions on Graphics*. 21(3), pp. 473–482, 2002.

[Kovar 2003] Kovar, L. and M. Gleicher, "Flexible Automatic Motion Blending with Registration Curves," ACM-Eurographics SCA'03, San Diego, CA, 2003.

[Laszlo 1996] Laszlo, J. F., M. Van de Panne, and E. Fiume, "Limit Cycle Control and its Application to the Animation of Balancing and Walking," *Proceedings of SIGGRAPH 96*, pp. 155–162, August 4–9, 1996.

[Laszlo 2000] Laszlo, J. F., M. Van de Panne, and E. Fiume, "Interactive Control For Physically-Based Animation," *Proceedings of SIGGRAPH 2000*, pp. 201–208.

[Lee 2002] Lee, J., J. Chai, P. Reitsma, J. K. Hodgins, and N. S. Pollard, "Interactive Control of Avatars Animated With Human Motion Data," *ACM Transactions on Graphics*. 21(3), pp. 491–500, 2002.

[Menache 2000] Menache A., *Understanding Motion Capture for Computer Animation and Video Games*, ISBN 0-12-490630-3, Morgan Kaufmann, 2000.

[Multon 1999] Multon F., L. France, M-P. Cani-Gascuel, and G. Debunne, "Computer Animation of Human Walking: a Survey," *The Journal of Visualization and Computer Animation,* vol. 10, pp. 39–54 (1999), J. Wiley & Sons.

[Murray 1967] Murray, P., "Gait as a total pattern of movement," *American Journal of Physical Medicine*, vol. 46, no. 1, pp. 290–333, 1967.

[Park 2002] Park, S. I., H. J. Shin, and S. Y. Shin, "On-line Locomotion Generation Based on Motion Blending," *Proceedings of ACM SCA'02*, pp. 105–111, ISBN:1-58113-573-4, 2002.

[Ponder 2003] Ponder, M., G. Papagiannakis, T. Molet, N. Magnenat-Thalmann, and D. Thalmann, "VHD++ Development Framework: Towards Extendible, Component Based VR/AR Simulation Engine Featuring Advanced Virtual Character Technologies," *Proceedings of Computer Graphics International* (CGI), IEEE Computer Society Press, 2003.

[Pullen 2002] Pullen, K. and C. Bregler, "Motion Capture Assisted Animation: Texturing and Synthesis." *ACM Transactions on Graphics.* 21(3), pp. 501–508, 2002.

[Reynolds 1999] Reynolds, C. W., "Steering Behaviors For Autonomous Characters," *Proceedings of Game Developers Conference 1999*, San Jose, CA. Miller Freeman Game Group, San Francisco, pp. 763–782, 1999.

[Shin 2001] Shin, H. J., L. Lee, and S. Y. Shin, "Computer Puppetry: an Importance-Based Approach," *ACM Trans. On Graphics,* vol. 20 (2), pp. 67–94, ISSN:0730-0301, 2001.

[Sun 2001] Sun, H. and D. Metaxas, "Automating Gait Generation," *Proceedings of SIGGRAPH 2001 Conference*, Los Angeles, CA, August 12–17, 2001.

[Ulicny 2002] Ulicny, B. and D. Thalmann, "Towards Interactive Real-Time Crowd Behavior Simulation," *Computer Graphics Forum,* vol. 21 (4), pp. 767–775, December 2002.

[Van de Panne 1997] Van de Panne, M., "From Footprints to Animation," *Computer Graphics Forum,* vol. 16 (4), pp. 211–223, 1997.

[White 1990] White, A. A. and M. M. Punjabi, *Clinical Biomechanics of the Spine*, 2d ed., J. B. Lippincott Company, Philadelphia, PA ISBN 0-397-50720-8, 1990.

An Architecture for Integrating Plan-Based Behavior Generation with Interactive Game Environments

R. Michael Young, Mark O. Riedl, Mark Branly, Arnav Jhala,
North Carolina State University
Raleigh, NC 27695
young@csc.ncsu.edu

R. J. Martin
Knowwonder Digital Mediaworks
rockdeity@mindspring.com

C. J. Saretto
Microsoft Corporation
cjsar@microsoft.com

Abstract

One central challenge for game developers is the need to create compelling behavior for a game's characters and objects. Most approaches to behavior generation have either used scripting or finite-state approaches. Both of these approaches are restricted by the requirement to anticipate game state at design time. Research in artificial intelligence (AI) has developed a number of techniques for automatic plan generation that create novel action sequences that are highly sensitive to runtime context.

In this article, we describe an architecture called Mimesis, designed to integrate a range of intelligent components with conventional game engines. The architecture is designed to bridge the gap between game engine design and development and much of the work in AI that focuses on the automatic creation of novel and effective action sequences. Users of the system construct two parallel models of the game world, one using extensions to existing game engine code, the other using techniques for explicit modeling of actions in terms of their requirements for execution and their effects on the game world.

When integrated with a game engine, Mimesis acts as a runtime behavior generator, responsible for both generating *plans*—coherent action sequences that achieve a specific set of in-game goals—and maintaining the coherence of those plans as they execute in the face of unanticipated user activity. In this article, we describe the architecture, its main components, and the APIs available for integrating it into existing or new game engines.

INTRODUCTION

One central challenge for game developers is the need to create compelling behavior for a game's characters and objects. For example, in simulation games, this challenge involves the accurate mathematical modeling of the rules of the real world being modeled (e.g., the creation of a physics engine). For many genres, however, engaging behavioral control is less prescribed, requiring the developer to build code that, hopefully, creates highly context-sensitive, dynamic behaviors for its system-controlled characters. Most approaches to behavior generation have either involved the use of precompiled scripts or the development of control software that implements a finite-state machine. While scripting approaches may require less complicated design than that needed to construct a finite-state behavioral model, a scripting language's runtime requirements typically limit the behaviors of the characters that it controls to be less sensitive to context (e.g., all actions in a script's sequence are fixed at design time and cannot readily be adapted should the game context change or the script be run more than once). In contrast, finite-state approaches can provide characters with a wider range of behaviors, but require the programmer to anticipate the full range of expected states and the transitions between them [Pottinger 2003]. These states and transitions, enumerated at design time, cannot be readily modified once a game is built.

Research in AI has developed a number of techniques for automatic plan generation that create novel action sequences that are highly sensitive to runtime context. Recent work [Cavazza 2002; Hill 2003] has sought to integrate this work with game environments in order to create intelligent, reactive characters, especially in the context of interactive narrative. In this article, we describe an architecture called Mimesis, designed to integrate a range of special-purpose intelligent components with conventional game engines. We describe the architecture, several of its main components and the APIs available for integrating it into existing or new game engines.

The Mimesis system is currently being used by the Liquid Narrative Group at North Carolina State University to construct interactive narrative-oriented games. The architecture is specifically designed to bridge the gap between game engine design/development and much of the work in artificial intelligence that focuses on the automatic creation of novel and effective action sequences. Users of the system construct two parallel models of the game world, one using extensions to existing game engine code, the other using techniques for explicit modeling of actions in terms of their requirements for execution and their effects on the game world. When integrated with a game engine, Mimesis acts as a runtime behavior generator, responsible for both generating *plans*—coherent action sequences that achieve a specific set of in-game goals—and maintaining the coherence of those plans as they execute in the face of unanticipated user activity.

The process of constructing a plan involves a number of specialized functions, including reasoning about the actions of individual characters, generating any character dialog or narration to be provided by the system, creating cinematic camera control directives to convey the action that will unfold in the story, and so forth. To facilitate the integration of corresponding special-purpose reasoning components, the Mimesis architecture is highly modular. Individual components within the Mimesis run as distinct processes (typically on distinct processors, although this is not a requirement); components communicate with one another via a well-defined socket-based message-passing protocol; developers extending Mimesis to provide new functionality wrap their code within a message-passing shell that requires only a minimal amount of customization.

While game engines are well-suited for building compelling interactive game worlds, the representation that they use to model game worlds does not match well with models typically used by AI researchers. The internal representation of most game engines is *procedural*—it does not use any formal model of the characters, setting, or the actions of the stories that take place within it. In contrast, most intelligent interfaces provide explicit *declarative* models of action, change, and the knowledge used to reason about them. Consequently, direct integration of intelligent software components with a game engine is not straightforward. To facilitate the integration of approaches that use these disparate representations, Mimesis augments a game engine's default mechanisms for controlling its virtual environment, using instead a client/server architecture in which low-level control of the game environment is performed by a customized version of the game engine (called the MWorld), and high-level reasoning about plan structure and user interaction is performed externally by a number of intelligent control elements.

The remainder of this article is organized as follows. In the next section, we summarize work integrating AI research and development with computer game engines. Then, we describe the Mimesis architecture further. Following that, we discuss the techniques used in Mimesis to integrate the procedural representation of game engines with the declarative representations used in AI systems. We also describe the techniques used by the planning components of Mimesis to create plans for controlling action within a game engine and how we monitor those plans to maintain their coherence effectively during execution. Next, we give a high-level discussion of the methodology for developing games that use the Mimesis architecture. Finally, we summarize the work described here and discuss our future work extending and employing the Mimesis architecture.

Related Work

Work that integrates intelligent reasoning capabilities with existing game engines can be roughly categorized into three groups based on the degree to which specific AI and game-engine elements are linked in their design. The most prevalent approach is to develop systems in which the AI and game engine are *mutually specific*. In these systems, the focus has been on creating new functionality within a specific game engine using a specific set of intelligent reasoning tools. For example, several researchers have explored the use of planning algorithms similar to the ones employed within Mimesis to generate novel action sequences—that is, action sequences that have not been pre-scripted or otherwise defined ahead of time—for characters inside existing game engines. For example, [Cavazza 2002; Charles 2003] have integrated both hierarchical task planning and heuristic search planning with *Unreal Tournament* to explore the creation of interactive narrative within games. However, they have not yet extended their work to generate API across a range of development environments. The work on the Mission Rehearsal Exercise by Hill, et al [Hill 1995] integrates an intelligent system based on the Soar cognitive architecture with a sophisticated virtual reality training environment. This system, too, is specific to the high-end runtime environment used to create the training modules.

A second category of systems integrating AI techniques with game engines can be defined as *AI specific*. In these approaches, a specific collection of AI tools has been designed or adapted for use across more than one game environment. For example, work by Laird and his students [Laird 2001] has integrated the Soar architecture mentioned previously into games such as *Quake* and

Decent 3. In this work, Soar models the cognitive state and reasoning performed by individual agents acting within the game world; global coherence of action is limited to that which emerges based on loosely coordinated individual action. The work by Atkins, et al [Atkins 1998] on the Hierarchical Agent Control system has been integrated with real-time strategy games, although the system has specifically been designed for use with a range of agent environments. Because their approach is quite general, they assume no direct access to the internal state of the runtime environment. Instead, action sequences must include specific sensing actions, computation that is often unnecessary when a game environment is generating actions for itself.

The third category of systems can be defined as *game specific*; that is, systems whose design goals include the ability for users to provide customized AI elements for integration into a specific game engine. For example, the Gamebots project, developed jointly at ISI and CMU [Adobbati 2001], is a game-specific architecture that defines a socket-based interface allowing intelligent agents to control bots within *Unreal Tournament*. While the API developed by this project allows easy integration of external code with UT, no facilities for generating or controlling the execution of action sequences is provided.

The Mimesis Architecture

Design Overview

In this article, we describe two specific functions that Mimesis provides when integrated with a conventional game engine:

- The generation of intelligent, plan-based character/system behavior at runtime.
- The automatic execution-monitoring and response generation within the context of the plans that it creates.

This article focuses on the former rather than the latter, although execution monitoring and replanning are discussed in the section *Mediation*.

To provide this functionality, Mimesis addresses three main design challenges. First, it provides a well-defined bridge between the disparate representations used by game engines and AI software. The architecture specifies an action representation for use by AI components with well-understood syntax and semantics and a methodology for creating game-side code that preserves that semantics. Second, it provides an API for game developers that can be readily integrated in a typical game engine design. This API has been used to construct a range of applications using both custom-built and commercial game engines. Finally, its architecture facilitates the integration of new intelligent modules, allowing researchers to extend the functionality that Mimesis can provide game developers. The architecture is component-based; individual components register themselves at system startup and communicate via socket-based message passing, using a set of predefined XML message types. This approach facilitates the use of a collection of special-purpose processes that can be easily extended.

Work to date on the Mimesis architecture has been performed in the context of research integrating theories of interactive narrative with computer game development [Christian 2003; Riedl 2003; Young 1999; Young 2003]. Consequently, much of the terminology used in this paper is related to narrative structure. Those terms containing references to story refer primarily to action taking place within the game world. This includes both the action of the game's characters as well as the behavior of inanimate objects such as doors, weapons, vehicles, and so forth. Those terms referring to *discourse* refer primarily to the media resources available within the game engine to tell the story. This includes items such as a 3D camera, narration, and background music. While the terminology is focused on narrative elements, the architecture itself can be applied to a range of game types (Figure 3.1).

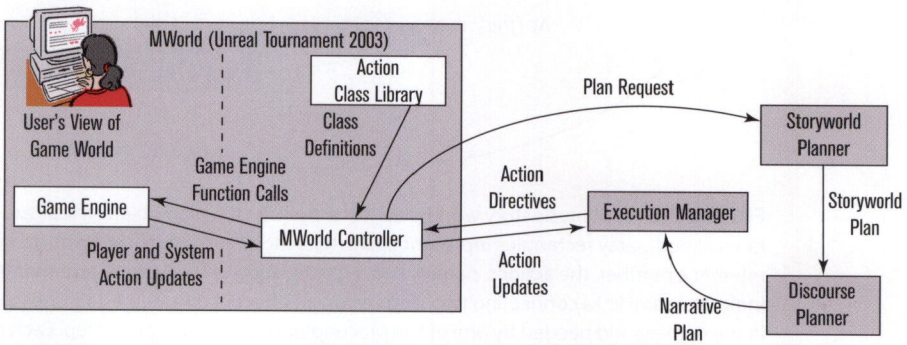


FIGURE 3.1 The Mimesis system architecture shown with an MWorld built using *Unreal Tournament 2003* as a sample game engine. Individual gray boxes indicate components described in *Components*. Within the MWorld component, the vertical dashed line represents the boundary between code created by the Mimesis developer (to the right of the line) and that created by the game engine developer (to the left of the line).

When integrated with a game engine, activity within Mimesis is initiated by a *plan request* made by the game engine itself. To make a plan request, the game engine sends a message to the component called the *story world planner*. The content of a plan request identifies a specific story problem that needs to be solved; for example, what goals the story actions must achieve and what library of actions are available to build the action sequence from (more detail about the nature of plan goals used by the story world planner is provided in the section *Components*). In response, the system a) creates a plan to control action within the game designed to achieve the story's goals, b) executes the plan's actions, and c) monitors their execution, adapting the plan should one or more of the actions fail to execute correctly.

To create and execute a plan, the Mimesis components follow the process outlined in Figure 3.2. When the story world planner receives a plan request, it creates a *story world plan*, a data structure that defines an action sequence for the relevant characters and other system-controlled objects within the game world. The story world planner passes this plan to the *discourse planner*, a component responsible for building a *discourse* plan, a structure that controls the camera, background music, and other media resources within the game world during the execution of the story world plan.

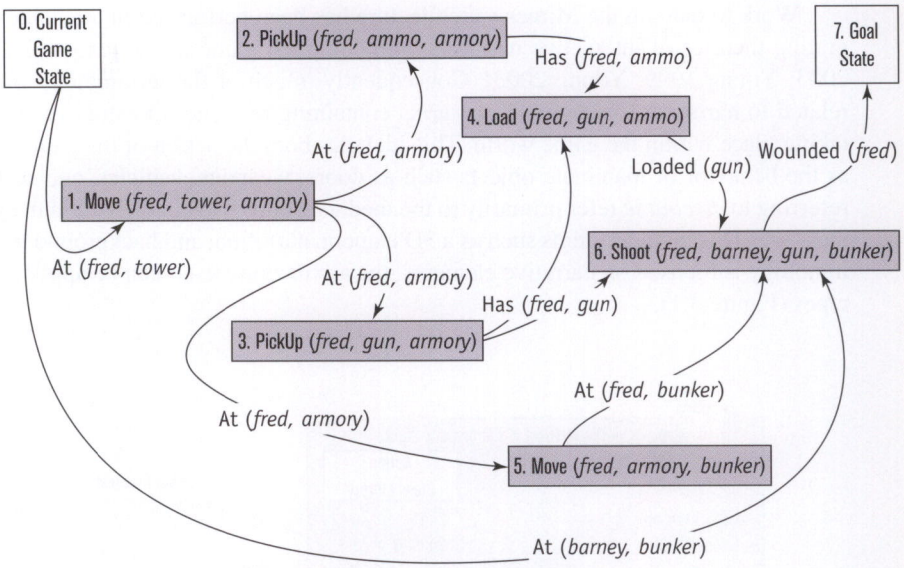


FIGURE 3.2 A Mimesis story world plan (for simplicity, the plan's hierarchical structure has been elided). Gray rectangles represent character actions and are labeled with an integer reference number, the actions' names, and a specification of the actions' arguments. Arrows indicate causal links connecting two steps when an effect of one step establishes a condition in the game world needed by one of the preconditions of a subsequent step. Each causal link is labeled with the relevant world state condition. Temporal ordering is roughly indicated by left-to-right spatial ordering. The white box in the upper left indicates the game's current state description, and the box in the upper right indicates the current planning problem's goal description. This plan involves one nonplayer character, Fred, moving from a tower to the armory (Step 1), picking up some ammo (Step 2) and a gun (Step 3), loading the gun (Step 4), and moving to the bunker (Step 5). There, Fred uses the gun to shoot another character, Barney, wounding him (Step 6).

The discourse planner synchronizes the communicative actions of the discourse plan with all character and environmental actions of the story world plan, creating an integrated *narrative plan* describing all system, nonplayer character, and user activity that will execute in response to the game engine's plan request. The narrative plan is sent to the *execution manager*, which builds a directed acyclic graph (DAG) representing the temporal dependencies between the execution of each action within the narrative plan. The execution manager then acts as a process scheduler, iterating through a cycle of a) selecting the minimal elements in the DAG (that is, those actions that are currently ready to begin execution), b) sending commands to the game engine to execute those actions, and c) receiving updates from the game indicating that actions have successfully (or unsuccessfully) completed their execution.

Components

Mimesis uses five core components, which we describe in more detail in this section. The *mimesis controller* (MC) acts as the system's central registry; upon initialization, each component connects to the MC via a socket connection and identifies the set of messages that it can accept. Once all components have registered, the MC serves as a message router. Components create messages with specific message types and send them via a socket connection to the MC. The MC forwards each message, again via socket connection, to the appropriate handler component(s) based on the registration information each component has provided.

The *story planner* is responsible for handling plan requests initiated by the game engine. A plan request contains three elements. First, it contains an encoding of all relevant aspects of the current game state. This encoding is managed by special-purpose functions that translate values stored in the game data structures into tuples (e.g., first-order logic atomic sentences). Second, it names one of a set of predefined libraries of actions that can be used by the story planner to compose action sequences. Finally, it contains a set of goals for the plan; that is, a listing of conditions in the game that must be true at the time that the plan ends its execution. These goals are also defined using the same tuple language that is used to characterize the current game state. An example entry from an action library is shown in Figure 3.3. Tuples appearing in the current state description and the story world plan's goals might include the following:

Current State Description (partial)	Story World Plan Goals
...	...
(Holding Barney Blaster_001)	(Health-level Fred 0)
(Ammo-count Blaster_001 20)	(In-inventory Barney KeyCard_032)
(Enemy Barney Fred)	(In-inventory Barney Crystal_001)
(Health-level Barney 95)	(Destroyed Klingon-Cruiser_01)
(Connects Bridge Galley Door_2)	
(Not (Locked Door_2))	
(Controls KeyCard_032 Door_2)	
(Human Barney)	
(Klingon Fred)	

The story planner responds to the plan request by creating a *story world plan*, a data structure that specifies the actions of the characters in the game and the system-controlled objects that will execute over time to achieve the plan request's goals. The section *Mimesis Control Model* discusses our approach to plan creation in more detail; however, a complete review of the type of planning used in Mimesis is beyond the scope of this article. Readers should see [Weld 1994] for a more thorough introduction.

To create the plan, the planner composes sequences of actions drawn from the action library specified in the plan request, searching for an action sequence that will lead from the current game state to one in which all of the plan's goals are met. By creating behavioral control on demand, a

planning approach has several advantages over those techniques that pre-script behavior. First, a planning approach can create action sequences that will successfully execute starting in complex game states that may not be anticipated by developers at design time. Second, because planners search through a space of possible plans while constructing the story plan, heuristics can be used to guide this search so that the nature of the final plan can be tailored to individual users' preferences. For example, should a player prefer stealthy opponents, plans that use covert movement and stealth attacks can be selected over ones that employ frontal attacks. These types of preferences can be acquired by settings explicitly set by each user at startup or stored in a profile. (Future work will investigate the acquisition of these preferences by tracking the user's actions in similar contexts and exploring the use of learning algorithms.) Third, as we discuss briefly in the section *Execution Management, Monitoring, and Mediation*, planners use techniques to create their plans that add explicit models of the causal and temporal relationships between their plans' actions. Analysis of these structures facilitates replanning in the face of action failure as well as the generation of effective explanations for story action, similar to the techniques used by intelligent tutoring systems to explain their own instructional plans [Swarout 1991].

Once the story planner has created the story world plan, it sends the plan to the *discourse planner*, along with a library of communicative actions that can be used by the game engine to convey the unfolding action of the story. These actions might include directives for a 3D camera controller, narrative voice-overs, or the use of background music. The discourse planner creates an action sequence containing actions to be carried out not by characters in the story world but by the game engine's media resources. Because these actions must be executed concurrently with the story plan itself, the discourse planner integrates the two plans, creating a *narrative plan* that contains explicit ordering relationships between actions from the two.

Depending on the plan request issued by the game engine, the discourse plan may be as transparent as a sequence of player-controlled first-person camera shots or as complex as a sequence of pans, tracks, and other cinematic techniques (for cut scenes or fly-throughs). Like the story planner, the discourse planner uses special-purpose knowledge to construct its plans. For example, knowledge encoding the use of film idioms and contextual information about the current level's scene geometry can be used to construct cinematic camera plans tailored to the layout of a given game map and the current location of players and characters involved in the action. Because of space limitations, the remainder of this article focuses on the use of story world plans rather than discourse plans. [Young 1994] provides a discussion of Longbow, the discourse planner used in Mimesis. Bares [Bares 1998] and Christianson, et al [Christianson 1996] provide further discussion on automatic camera control in 3D worlds.

The discourse planner sends the narrative plan to the *execution manager*, the component responsible for driving the story's action. The execution manager builds a DAG whose nodes represent individual actions in the plan and whose arcs define temporal constraints between actions' orderings. The execution manager iteratively removes an action node from the front of the DAG, sends a message to the game engine that initiates the action's execution, and updates the DAG to reflect the state of all actions that are currently running or have recently completed execution.

The *MWorld* includes three elements: the game engine, the code controlling communication between the MWorld and the other Mimesis components, and the a*ction classes,* class definitions

that specify the behaviors of each action operator represented within the story and discourse planners (Mimesis' dual approach to action representation is discussed further later in the article). When the MWorld receives a message from the execution manager directing an individual action to execute within the game world, it extracts the string name of the action from the message and maps the name onto a specific action class. This mapping is done via conventions used in the naming of the actions in both the operator library used by the planners and in the action class hierarchy used by the MWorld. In a similar manner, the MWorld extracts the string identifiers of each of the action's arguments and maps them onto game world objects. This mapping is done, however, by accessing a lookup table created and maintained by the MWorld. Game code registers object references in this table, along with their unique string identifiers, and the MWorld maps objects into an action instance's actual arguments based on positional ordering in the message arriving from the execution manager. From this mapping, an instance of the action class is created, the action's arguments are passed to it, and the action's default execution method is called. When each action halts its execution, it notifies the execution manager of its successful (or unsuccessful) completion.

In addition to the five core components described previously, Mimesis can be configured with additional components to provide extended functionality. For example, components that provide SQL database access, natural language generation capabilities [Elhadad 1992 & Young 1994], and HTTP services have been used in our prototype applications. The component architecture facilitates the integration of these and other new modules into the framework. APIs for constructing new components are available for a range of environments, including Java 2SE, C#, C++, and Allegro Common Lisp.

Mimesis Control Model

The Mimesis Dual Action Representation

Mimesis brings together two representations for action: the procedural representations used by game engines and the declarative representations used by AI systems such as planners. Each of these representations has individual strengths—the efficient management of game state by game engine code and the ability to reason explicitly about action and state change by planning systems. In the following sections, we describe how Mimesis attempts to link the two in a way that preserves the advantages of both.

Model-Based Action Generation

Declarative representations of actions typically used by AI systems characterize the properties of actions—for example, under what circumstances an action can be executed and how the action alters the world state—without explicitly stating *how* the action performs its tasks. The planners in Mimesis use a declarative representation in which an action is represented using two main elements: its *preconditions* and its *effects*. An action's preconditions are a set of predicates describing those conditions of the game world that must hold for the action to execute correctly. An action's effects are a set of predicates capturing all changes to the world state made by the action once it

successfully executes. Figure 3.3 shows an example plan operator for the action of one character shooting another with a weapon[1].

Operator Shoot (?shooter ?target ?weapon ?room)

Constraints:
(health-level ?target ?t_health)
(damage-level ?weapon ?damage_amount)
Preconditions:
(has-weapon ?shooter ?weapon)
(has-ammo ?weapon)
(in-room ?shooter ?room)
(in-room ?target ?room)
Effects:
(health-level ?target
(- ?t_health ?damage_amount))

FIGURE 3.3 A DPOCL plan operator for the shoot action used in the plan in Figure 3.2. In this operator, the Constraints section is used to provide bindings between the relevant game world objects and the operator's local variables. The Preconditions ensure that a) the character doing the shooting has the weapon being used to shoot, b) the weapon is loaded, c–d) the shooter and the target character are in the same room. The effects of the action specify that the health level of the target character is decremented by the damage inflicted by the weapon being used.

Mimesis uses DPOCL [Young 1994] as the planning algorithm for story planning. A DPOCL plan contains elements composed from five central types. First, they contain *steps* representing the plan's actions. *Ordering constraints* define a partial temporal order over the steps in a DPOCL plan, indicating the order in which the steps must be executed. Hierarchical structure in a DPOCL plan is represented by *decomposition links*: a decomposition link connects an abstract step to each of the steps in its immediate subplan. (For expository purposes, the examples in this section have been structured to eliminate the need for hierarchical planning; only casual planning is performed. The DPOCL algorithm, however, along with the plan-based techniques that we present here, are applicable to planning problems using more expressive representations (i.e., casual as well as decompositional structure). Finally, DPOCL plans contain *causal links* between pairs of steps. A causal link connects one step to another just when the first step has an effect that is used in the plan to establish a precondition of the second step.

[1] Additional elements can also be added to plan representations. For instance, the operator in Figure 3.3 uses additional category of constraints, used to query the world state to obtain state variable values for use in the context of the operator.

DPOCL uses *refinement search* [Kambhampati 1995] as a model for its plan reasoning process. Refinement search is a general characterization of the planning process as search through a space of plans. A refinement planning algorithm represents the space of plans that it searches using a directed graph; each node in the graph is a (possibly partial) plan. An arc from one node to the next indicates that the second node is a refinement of the first (that is, the plan associated with the second node is constructed by repairing some flaw present in the plan associated with the first node). In typical refinement search algorithms, the root node of the plan space graph is the empty plan containing just the initial state description and the list of goals that together specify the planning problem. Nodes in the interior of the graph correspond to partial plans, and leaf nodes in the graph are identified with completed plans (solutions to the planning problem) or plans that cannot be further refined due to, for example, inconsistencies within the plans that the algorithm cannot resolve. In Mimesis, the initial planning problem for DPOCL is created using the specifications of the current and goal states taken from the game engine's plan request. The approach to plan generation as search facilitates the creation of plans tailored not just to the particular state of the game world at planning time, but to preferences for certain types of action structure. Search control rules can be defined that direct search toward (or away from) plans that use certain objects, tools, routes, characters, or types of action.

Procedural Representations for Action

To ensure that every step in a plan can be executed by the game engine, the game developer must create one action class for every action operator in the plan library. The implementation of each action class is responsible for preserving the semantics of the action operator defined in the planner's action library. To this end, the MWorld's abstract action class defines four functions, three that the game developer must provide definitions for in the action classes that he or she defines. An action's `CheckPreconds()` function is responsible for verifying that the conditions described in the corresponding operator's preconditions currently hold in the game world. The `Body()` function is responsible for changing the state of the world in accordance with the meaning of the action operator. The `CheckEffects()` function verifies that the conditions described in the operator's effects have actually been obtained in the game world immediately after its execution.

The `Executing()` function is an abstract function defined only in the parent action class. This function, shared by all action classes, first calls the action's `CheckPreconds()`. If one of the action's preconditions is not met, the `Executing()` function stops execution and sends a failure message to the execution manager; the game engine and the story world planner then synchronize their views of the current game state, and replan the story's action. Otherwise, the function calls the action's `Body()` and then calls the action's `CheckEffects()` function. If one of the action's effects does not hold, the function halts execution and reports this condition to the execution manager. Otherwise, if no problems were encountered, the `Executing()` function reports that the action has completed successfully. An example action class definition is shown in Figure 3.4. This definition is written in UnrealScript, *Unreal® Tournament*'s scripting language, although APIs exist for a range of languages and are discussed in the section *Building Games Using Mimesis*.

```
class Shoot extends Action;

var MController Agent;
var Pawn MyTarget;
var Weapon MyWeapon
var int OriginalHealth;
var int precondResult;
var int effectsResult;

function int CheckPreconds(){
        if (Agent.Pawn.Weapon != MyWeapon)
                {return 0;}
        else if (!(Agent.Pawn.Weapon.HasAmmo()))
                {return 1;}
        else if (Agent.Room != MyTarget.Room)
                {return 2;}
        else
                {return -1;}
}

function int CheckEffects(){
        if( (MyTarget.Health >= OriginalHealth))
                {return 0;}
        else {return -1;}

function void Body(){
        local int fireMode;
        OriginalHealth = MyTarget.Health;
        Agent.Pawn.SetPhysics(PHYS_None);
        Agent.Pawn.SetViewRotation(rotator(MyTarget.Location -
                Agent.Pawn.Location));
        fireMode = Agent.Pawn.Weapon.BestMode();
        Agent.Pawn.Weapon.StartFire(fireMode);
}

state Executing {

Begin:
        precondResult = CheckPreconds();
        if (precondResult != -1) {
                reportPrecondFailure(precondResult);
                gotoState('Idle'); }
        Body();
        effectsResult = CheckEffects();
        if (effectsResult != -1) {
                reportEffectFailure(effectsResult);
                gotoState('Idle'); }
        reportActionSuccess();
```

CheckPreconds() checks each precondition from the corresponding operator in the order in which they are defined there. When a precondition does not hold in the current game state, an integer identifying the failed precondition is returned. If all preconditions hold, the function returns -1

CheckEffects() runs after the body of the action completes. It verifies each of the operator's effects. When an effect does not hold in the current game state, an integer identifying the failed effect is returned. If all effects hold, the function returns -1.

Body() implements the operator's behavior, changing the game state according to the intended meaning of the operator.

The Executing State is identical across all action classes, and so is typically defined in the top-level Action class. It is included here for reference.

The Executing code first checks the action's preconditions. If all preconditions are met, then it runs the action's body. Next, it checks the action's effects. If all effects hold, then the action sends a message to the execution manager indicating that it has completed successfully. If an error is encountered along the way, the action sends an error report to the execution manager, facilitating replanning.

FIGURE 3.4 An example UnrealScript action class for the Shoot operator defined in Figure 3.3.

Execution Management

To control and monitor the order of execution for the actions within the narrative plan, the execution manager builds a DAG that represents all temporal dependencies between the actions in the DAG. This temporal information is created by the story world and discourse planners when the plans are first built and extracted by the execution manager when it receives the narrative plan. An example execution DAG for the plan from Figure 3.2 prior to any plan execution is shown in Figure 3.5a.

To initiate the execution of an action from the execution DAG, the execution manager sends a message to the MWorld specifying the action's name and a tuple naming the action's arguments.

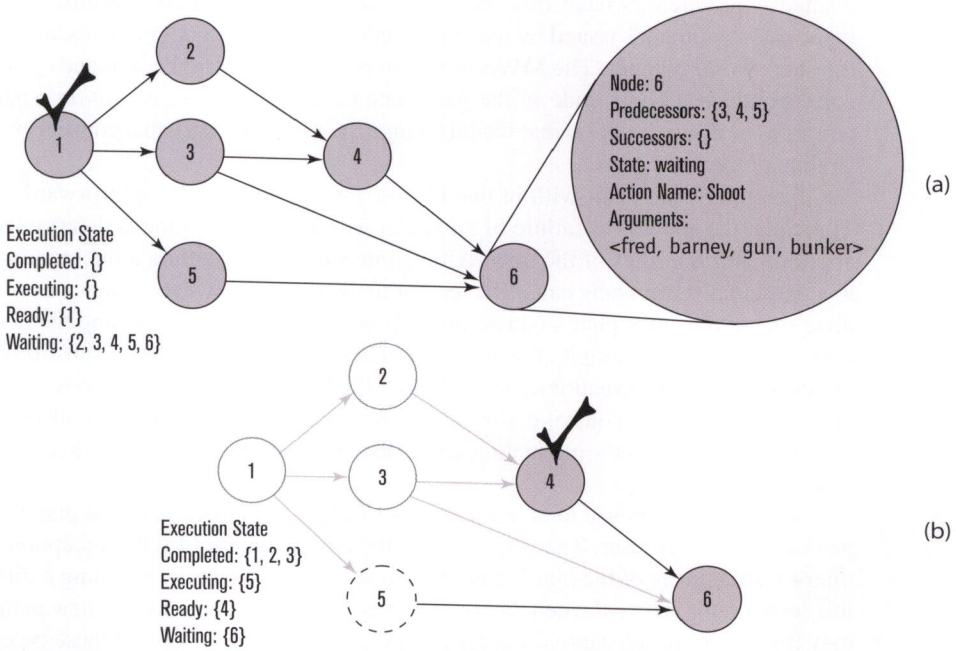


FIGURE 3.5 Two examples of the execution DAG used by the execution manager, matching the plan shown in Figure 3.2. In these figures, gray circles indicate steps in the DAG that are as yet unexecuted. An arc between two nodes indicates that the first node must complete execution before the second node can begin execution. Nodes with grayed borders are ones that have completed execution; those with dashed borders indicate actions that are currently executing. Nodes with check marks are currently ready to execute. In Figure 3.5a, execution of the plan has not yet begun. Figure 3.5b shows the execution DAG after Fred has gone to the armory, picked up the ammo and gun, and is in the process of moving to the bunker. At this point, he's ready to load the gun, but has not yet done so.

To build this message, the narrative plan's identifiers for the action and its parameters are used to create string names that act as unique identifiers agreed upon by both the MWorld developer and the developer of the plan operators (the string names are typically created based on the symbol names used by the plan operators). As mentioned previously, the MWorld uses a registration procedure to map these names into the actual arguments of function calls for action class instances.

Mediation

Complications to the plan execution process arise when the user performs actions within the story world that interfere with the structure of the story plan. For example, suppose that the user controls a character named Betty in the world of the plan in Figure 3.2. If the user decides to pick up the ammo in the armory before Fred moves there, Fred's subsequent pickup action would fail, as would each subsequent action that depended on the ammo being available to Fred. Because *every* action that the user performs might potentially change the world state in a manner that could invalidate some as-yet-unexecuted portion of the narrative plan, the MWorld checks for such unwanted consequences each time the player initiates a potentially harmful action. The MWorld maps such commands issued by the player onto the same plan operators used to create the plans by the story world planner. The MWorld monitors the user's action commands prior to executing the corresponding action's code in the game engine and signals an *exception* whenever the user attempts to perform an action that would change the world in a way that conflicts with the causal constraints of the story plan.

Exceptions are dealt with in one of two ways. The most straightforward is via *intervention*. Typically, the success or failure of an action within a game engine is determined by the code that approximates the rules of the underlying game world (e.g., setting a nuclear reactor's control dial to a particular setting may cause the reactor to overload). However, when a user's action would violate one of the story plan's constraints, Mimesis can intervene, causing the action to fail to execute. In the reactor example, this might be achieved by surreptitiously substituting an alternate set of action effects for execution, one in which the "natural" outcome is consistent with the existing plan's constraints. A control dial momentarily jamming, for example, will preserve the apparent consistency of the user's interaction while also maintaining safe energy levels in the story world's reactor system.

The second response to an exception is to adjust the structure of the plan to *accommodate* the new activity of the user. The resolution of the conflict caused by the exception may involve only minor restructuring of the plan's causal connections; for example, selecting a different but compatible location for an event when the user takes an unexpected turn down a new path. Accommodation may involve more substantive changes to the story plan, however, and these types of modifications can be computationally expensive. For example, should a user unintentionally destroy a device required to rescue a game's central character, considerable replanning will be required on the part of the story and discourse planners.

To detect and respond to exceptions during the execution of a story world plan, the execution manager analyzes each plan prior to execution, looking for points where enabled user actions (that is, actions whose preconditions for execution are satisfied at that point in the plan) can threaten its plan structure. When potential exceptions are identified, the planner weighs the computational cost of replanning required by accommodation against the potential cost incurred when intervention breaks the user's sense of agency in the virtual world. A more detailed discussion of the interaction between the MWorld and the story planner with respect to exception detection and handling can be found in [Riedl 2003].

We have created two sets of APIs for use by developers integrating Mimesis functionality into their games. The first API, the *mclib*, is a library of functions that provides registration, message passing, and parsing functionality for new Mimesis components. APIs are available for C++, Java 2 Standard Edition, and Allegro Common Lisp version 6.2. The second, called the *mwlib*, provides the functionality needed to integrate new or existing game engines with the rest of the Mimesis components. The mwlib includes functions that allow an MWorld to register with the MC, to record unique names for game objects (facilitating the translation between XML strings and action class method invocation) and to report the progress of action class execution (via the `ReportPrecondFailure()`, `ReportEffectFailure()` and `ReportActionSuccess()` functions described previously).

We have developed mwlib libraries for all the environments for which mclibs exist. Further, we have created mwlib APIs for *Unreal Tournament*, *Unreal Tournament 2003*, and two text-based virtual reality environments: [MudOS; LambdaMOO]. To see how these APIs may be put to use, consider the following example in which a university research team partners with a game development studio to create an educational game in which players move through a simulation of the Monterey Bay Aquarium, a real-world marine-science learning center. The university team seeks to integrate their research on intelligent tutoring systems into the 3D game engine used by the game developers; in this case, Epic Games' *Unreal Tournament 2003*. The game developers have created levels that correspond to the Aquarium's exhibits (see Figure 3.6) and have written UnrealScript

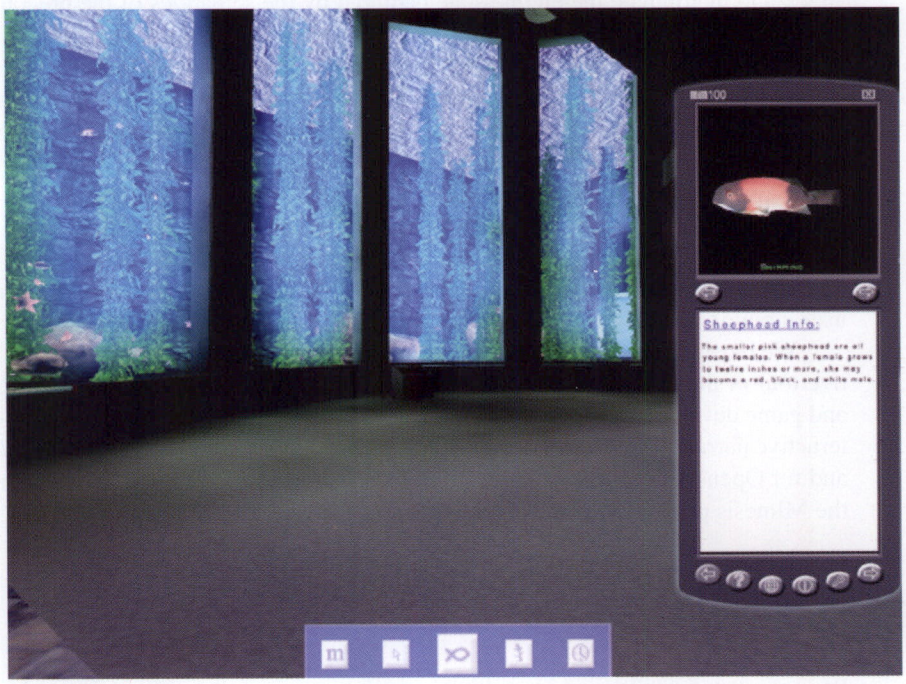


FIGURE 3.6 A screenshot of the Mimesis Virtual Aquarium, a simulation of a portion of the Monterey Bay Aquarium, a large marine-science learning center in Monterey, CA.

code to control both the Aquarium's inhabitants (i.e., marine life, visitors, and tour guides) and its physical environment (e.g., elevators, lights, TV monitors). The research group has developed effective pedagogical techniques for teaching material regarding the behaviors of marine animals that involve the tight coordination between animated pedagogical agents [Lester 1999] (e.g., tour guide characters), the fixed information resources available in the environment (e.g., the explanatory labels associated with each habitat in the aquarium), and the behavior of the animals themselves (e.g., the schooling behavior of tiger sharks before feeding). The team wants to integrate a planning system into the game to generate plans that control the tour guide and the marine animals living inside the Kelp Forest, one of the Aquarium's larger habitats. These plans will be customized according to a) queries about the environment posed by the player, and b) the state of the game world, including the position of the tour guide relative to the player, the composition of marine life present in the Kelp Forest at the time of the query, and the textual content of visible labels on the Kelp Forest window through which the player is looking.

To integrate the work of both project groups, the team first defines a library of plan operators using a GUI provided with the planning systems used by Mimesis. These plan operators specify the preconditions and effects for the primitive actions used within the planner to create story world plans. In the aquarium example, these might include actions such as gesturing at objects, speaking to the player, turning displays on and off (for tour guide characters), eating food, or retracting into one's shell and self-grooming (for animals such as sea turtles, snails, rockfish or otters).

Second, the team defines the set of UnrealScript action classes that will execute within the MWorld. One action class is defined for each action operator used by the planner; in each action, the `CheckPreconds()`, `Body()`, and `CheckEffects()` methods are written so that they correctly check and manipulate the engine state captured by the semantics of the preconditions and effects in the corresponding action operator. These methods will, in turn, typically call functions already defined in the developer's game-specific code or within the engine itself.

Third, the team writes initialization code that will register all of the game objects in the MWorld, allowing a mapping to be made from the names used to refer to these objects by the planner to the object instances in the game engine. The team extends the game's startup code to call the mwlib registration function to make the initial connection with the MC.

Finally, the team defines the points in the game where requests for plan structures will be made. Plan requests can be initiated—and the resulting plans executed—at the very beginning of the player's session, at fixed trigger points throughout gameplay, or dynamically in response to user input or arbitrary game engine code.

We have successfully used the process just described to create a prototype intelligent interactive tour of the Monterey Bay Aquarium (except that we have filled the roles of both researchers and game developers). Additional small-scale games demonstrating features of our research on interactive narrative have been developed using the same methodology for *Unreal Tournament 2003* and for OpenGL® worlds running on the PlayStation® 2 (PS2Linux). More details can be found at the Mimesis project Web site *http://mimesis.csc.ncsu.edu/*.

Summary and Conclusions

We are extending the work described here along several dimensions. First, the current mclib and mwlib APIs are being implemented in Java 2 Micro Edition to integrate with games running on

mobile platforms like PDAs and mobile telephones. We are considering creating mwlibs for other 3D game engines as well. The APIs and several of the existing components are being implemented within the .NET framework to increase platform coverage.

We are also exploring ways to extend the architecture to handle control of multiplayer games. Even though the components of the current system can run in a distributed manner, with each component executing on distinct processors, the runtime performance of the planning processes currently limits Mimesis' application to single-user games. To address this limitation, we are exploring the performance trade-offs involved in the execution of the architecture on a grid-based gaming platform [Levine 2003; Tapper 2002] and addition of inter-planner communication needed to coordinate plans for multiple players.

As mentioned earlier, Mimesis is being used in our research group to investigate new models of interactive narrative in games [Riedl 2003; Young 2003]. In this work, we are building new intelligent components for modeling the user's knowledge about the stage of the game world, for drawing inferences about a user's plans and goals based on observation of the user's game actions [Carberry 2001; Laird 2001], and for cinematic control of the game's camera [Bares 1998; Christianson 1996] for use, for example, in the automatic generation of cut-scenes or fly-throughs.

The central feature of the architecture is the effective integration of plan-based techniques with a range of game engines. Under the assumption that their representations of actions in the story world is correct, planners can compose action sequences that achieve story world goals in ways that vary according to context, providing new action sequences for each run of a given story and generating action sequences for new stories not originally defined by a game's designers. Further, assuming the accurate implementation of the semantics of the planner's operators within the game engine's native code, the plans can be shown to be provably sound; that is, they can be guaranteed to execute correctly as long as users do not interfere with their progress. Finally, the structure of the plans created by the planner contain temporal and causal annotations sufficient to detect when a user's actions will invalidate the plan, and these annotations guide replanning in ways that result in minimal effort on the part of the system.

While the architecture is now complete, evaluation of the system is ongoing. In particular, we are hoping to evaluate the computational effectiveness of the plan-based approach. A central aspect of the computational effectiveness of the system involves the time taken by the planning system to generate complex action sequences for story world plans. Currently, our test-bed scenarios involve libraries of 20–40 abstract actions (e.g., `rob-bank`, `open-vault`, `disarm-alarm`) and as many primitive actions (e.g., `pick-up`, `turn-to-face`, `move-to`, `shoot`, `swipe-cardkey`), and generate plans with roughly 50 primitive actions in under one second (running in Allegro Common Lisp on a Pentium 4 1GHz machine). While these plans are relatively short, they are adequate for many cut-scenes and other short action sequences.

We anticipate that the use of plan-generation techniques that interleave planning and execution will facilitate the effective scaling of our algorithms. In this approach, we will exploit the hierarchical structure of the plans used by Mimesis by a) first generating the complete plan down to the primitive level for just that portion of the plan that must execute first, and b) deferring the completion of the later portions of the plan until the initial portion of the plan is executing. In addition, we are reimplementing the DPOCL planning algorithm in C# to exploit the runtime efficiency of the .NET framework. Because the primary use of the planner comes at system startup (to generate the initial story world plan), the impact of the delay caused by the planning system can be folded in to the overall system initialization time, minimizing its impact on overall gameplay.

In summary, the Mimesis system defines an architecture for integrating intelligent plan-based control of action into interactive worlds created by conventional game engines. To this end, it bridges the gap between the representations used by game developers and those of AI researchers. It provides an API for game developers that can be readily integrated in a typical game engine design, and its component-based architecture makes the addition of new intelligent modules straightforward.

A detailed design document used for implementing new Mimesis components can be found on the Mimesis mclib page (*http://mimesis.csc.ncsu.edu/mclib*). All APIs described here are available for download from the Mimesis project home page (*http://mimesis.csc.ncsu.edu/*).

Acknowledgments

The work of the Liquid Narrative group has been supported by National Science Foundation CAREER award 0092586 and by Microsoft Research's University Grants Program. Contributions to the Mimesis architecture have been made by a number of students, including Dan Amerson, William Bares, Charles Callaway, D'Arcey Carol, David Christian, Shaun Kime, Milind Kulkarni, and the many students of CSC495, *Research Projects in Intelligent Interactive Entertainment*. Thanks also go to Epic Games for their support of our work.

References

[Adobbati 2001] Adobbati, R., A. N. Marshall, A. Scholer, S. Tejada, G. A. Kaminka, S. Schaffer, and C. Sollitto, Gamebots: A 3D virtual world test bed for multi-agent research, *Proceedings of the Second International Workshop on Infrastructure for Agents, MAS and Scaleable MAS*, Montreal, Canada, 2001.

[Atkins 1998] Atkins, M. S., D. L. Westbrook, P. R. Cohen, and G. D. Jorstad, AFS and HAC: Domain-General Agent Simulation and Control, *Working Notes of the Workshop on Software Tools for Developing Agents, National Conference for Artificial Intelligence,* pp. 89–95. 1998.

[Bares 1998] Bares, W., J. Gregoire, and J. Lester, "Realtime Constraint-Based Cinematography for Complex Interactive 3D Worlds," *Proceedings of the Conference on Innovative Applications of Artificial Intelligence*, 1998.

[Carberry 2001] Carberry, Sandra. "Techniques for Plan Recognition." *User Modeling and User-Adapted Interaction,* 11 (1–2), pp. 31–48, 2001.

[Cavazza 2002] Cavazza, M., F. Charles, and S. Mead, "Planning characters' behaviour in interactive storytelling." *Journal of Visualization and Computer Animation*, 13(2): pp. 121–131, 2002

[Charles 2003] Charles, F., M. Lozano, S. Mead, A. Bisquerra, and M. Cavazza, "Planning Formalisms and Authoring in Interactive Storytelling," *Proceedings of the First International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, 2003.

[Christian 2003] Christian, D. and R. M. Young, *Comparing Cognitive and Computational Models of Narrative Structure.* Liquid Narrative Technical Report TR03-001, Liquid Narrative Group, Department of Computer Science, North Carolina State University, Raleigh, NC. 2003.

[Christianson 1996] Christianson, D., S. Anderson, L. He, D. Salesin, S. Weld, and F. Cohen, "Declarative Camera Control for Automatic Cinematography," *Proceedings of the Conference of the American Association for Artificial Intelligence,* pp. 148–155, 1996.

[Elhadad 1992] Elhadad, M., *Using argumentation to control lexical choice: a unification-based implementation.* Ph.D. thesis, Computer Science Department, Columbia University, 1992.

[Hill 2003] Hill, R. W., J. Gratch, S. Marsella, J. Rickel, W. Swartout, and D. Traum, "Virtual Humans in the Mission Rehearsal Exercise System," *Kunstliche Intelligenz* (Special Issue on Embodied Conversational Agents), 2003.

[Kambhampati 1995] Kambhampati, S., C. A. Knoblock, and Y. Qiang, Planning as refinement search: a unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, 76: pp. 167–238, 1995.

[Laird] Laird, J. E., Using a computer game to develop advanced AI. *Computer*, July 2001, pp. 70–75.

[Laird 2001] Laird, J., "It knows what you're going to do: adding anticipation to a Quakebot," *Proceedings of the Fifth International Conference on Autonomous Agents*, pp. 385–392, 2001.

[LambdaMOO] LambdaMOO, LambdaMOO Community Page, available at *www.lambdamoo.info/*.

[Lester 1999] Lester, J., B. Stone, and G. Stelling, "Lifelike pedagogical agents for mixed-initiative constructivist learning environments," *International Journal of User Modeling and User-Adapted Interaction*, 9 (1–2), pp. 1–44, 1999.

[Levine 2003] Levine, D., M. Wirt, and B. Whitebook, *Advances in Computer Graphics and Game Development: Practical Grid Computing For Massively Multiplayer Games*, Charles River Media, 2003.

[Lewis 2002] Lewis, M., and L. Johnson, "Game Engines in Scientific Research," *Communications of the ACM*, pp. 27–31, vol. 45, no. 1, January 2002.

[Marsella 2003] Marsella, S., W. Johnson, and C. LaBore, "Interactive pedagogical drama for health interventions," *Proceedings of the 11th International Conference on Artificial Intelligence in Education*, Sydney, Australia, July 2003.

[MudOS] MudOS, *The Official MudOS Support Page*, available at *www.mudos.org*.

[Pottinger 2003] Pottinger, D., "Computer Player AI: The New Age," *Game Developer Magazine*, July, 2003.

[Riedl 2003] Riedl, M., C. J. Saretto, and R. M. Young, "Managing interaction between users and agents in a multi-agent storytelling environment," *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems*, June, 2003.

[Rosenbloom 2003] Rosenbloom, Andrew, "A game experience in every application," *Communication of the ACM*, pp. 29–31, vol. 46 no 7, July 2003.

[Seneff 1998] Seneff, S., E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue, "Galaxy-II: A Reference Architecture for Conversational System Development," *ICSLP '98*, pp. 931–934, Sydney, Australia, December, 1998.

[Swartout 1991] Swartout, W., C. Paris, and J. Moore, "Design for explainable expert systems." *IEEE Expert*, 6(3):58–64, 1991.

[Tapper 2002] Tapper, D., and S. Olhava, "IBM Global Services: Scoring Big on Utility-Based Gaming," *IDC Report #27453,* June 2002.

[van Lent 2001] van Lent, M., J. Laird, J. Buckman, J. Hartford, S. Houchard, K. Steinkraus, and R. Tedrake, "Intelligent Agents in Computer Games," *Proceedings of the National Conference of the American Association for Artificial Intelligence*, 2001.

[Weld 1994] Weld, D., "An Introduction to Least-Commitment Planning." *AI Magazine*, pp. 27–61, 1994.

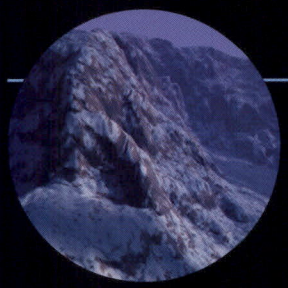
[Young 1999] Young, R. M., Notes on the Use of Plan Structures in the Creation of Interactive Plot. *Working Notes of the AAAI Fall Symposium on Narrative Intelligence*, Cape Cod, MA, 1999.

[Young 1994] Young, R. M., J. D. Moore, and M. Pollack, "Towards a Principled Representation of Discourse Plans," *Proceedings of the Sixteenth Conference of the Cognitive Science Society*, Atlanta, GA, 1994.

[Young 1994] Young, R. M., , Pollack, M. E., and Moore, J. D., "Decomposition and causality in partial-order planning," *Proceedings of the Second International Conference on AI and Planning Systems,* Chicago, IL, pages 188-193, July, 1994.

[Young 2003] Young, R. M., and M. Riedl, "Towards an Architecture for Intelligent Control of Narrative in Interactive Virtual Worlds." *Proceedings of the International Conference on Intelligent User Interfaces*, January, 2003.

Efficient Navigation Mesh Implementation



John C. O'Neill
Vicious Cycle Software, Inc.
joneill@viciouscycleinc.com

Abstract

The navigation mesh is a simple abstract representation of 3D world geometry that provides path-finding information to computer-controlled game objects. In addition, by modeling the navigation mesh to be mutually exclusive of the static obstacles in a game level (such as walls and other nonmoving structures), the information in a navigation mesh can be used to aid in collision avoidance. The navigation mesh provides information for movement within multiple triangles in the mesh by specifying directions through the edges between neighbor triangles. This article describes an efficient implementation to provide a solution to the path-finding problem, while taking into account memory storage and runtime requirements for use on current video game hardware, including the Xbox™, PlayStation® 2, and GameCube™. The approach can also be applied to PC applications with less restriction toward memory usage; it can be adapted for use in large-scale PC applications with greater memory allowances or used as described in this article. The in-game usage allows for containment of AI-controlled characters and navigation against the static environment with minimal overhead and computation time per object.

INTRODUCTION

A common feature that exists in virtually every computer or video game is the requirement for an "intelligent" computer-controlled game object, also referred to as an *avatar*, *entity*, or simply *AI*. One of the fundamental requirements of a successful AI is the ability to appear competent when navigating the virtual game world. The illusion of gaming intelligence is quickly dissolved away when the game player notices blatant errors in the computer's ability to navigate the same environment as his or her avatar. As the gaming world has evolved to visually rich and detailed 3D environments, the problem of path-finding and collision has escalated in complexity and computational requirements.

The use of a navigation mesh object (referred to as a *NavMesh* throughout this article) is an attempt to solve—or at least greatly simplify—the path-finding and static geometry avoidance problem. This approach does not attempt to replace dynamic object interaction and collision, but it will greatly reduce the need for dynamic object collision detection against the virtual physical barriers in a game level. The information that a NavMesh provides to the AI includes direction of movement toward a target point or object, collision avoidance, and valid playing field definition stored in what is referred to as a *connection map*. A valid playing field is simply the defined region that the game designer dictates the game AI should be allowed to move within and varies based on the game level requirements. Additionally, the storage structure for the vertices that define the triangles in a navigation mesh allows for 3D topology, meaning the same mesh can cross above or below triangles within the same NavMesh object. At Vicious Cycle Software, we have implemented the solution described within this article and have used NavMesh objects in a number of previous technology demos as well as two products currently in development for Take 2 Licensing: *Spy vs. Spy* and *Robotech: Invasion*. The specific usage in these two products will be described later in this article. The Vicious Cycle implementation of NavMeshes includes an improvement to existing solutions in the optimization of the path-finding algorithm. This is referred to as the *Iterative Constraint Rays* algorithm and provides a scalable solution to the accuracy of path-finding results based on the available CPU time.

Current implementations of NavMesh solutions exist that focus on near-optimal modeling of the game environment [Tozour 2002] and NavMesh object generation [White; Christensen 2002]. The solution described in this article focuses on advances in optimizations for processor utilization and memory requirements (see Figure 4.1).

Problem Description

The NavMesh implementation described in this article attempts to solve two main problems: navigation (path-finding) within the defined game level and static collision avoidance. Static geometry consists of solid, nondynamic mesh data (such as walls and other physical game geometry that contains collision information and does not move) or height-mapped terrain data. Path-finding involves the use of a series of edge connections to find a path from one triangle to another within the same NavMesh. The connections are stored as edge indices between triangles that lead from a source triangle to a destination triangle. The collision avoidance properties of the NavMesh movement algorithms provide an alternative solution to computing AI entity collision against static geometry. Typically, using traditional collision detection algorithms [Kaiser 2000], this calculation

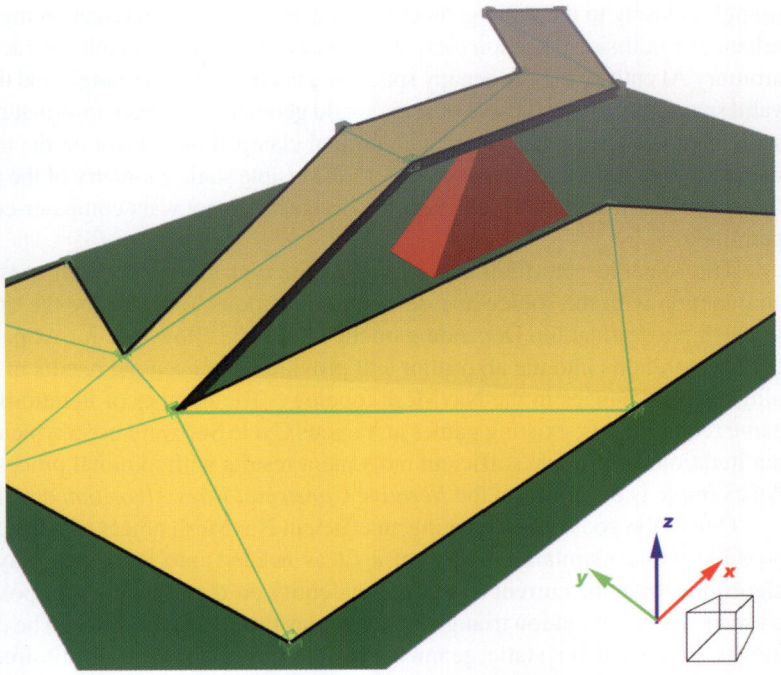


FIGURE 4.1 Basic navigation mesh showing 3D movement bounds.

can be time consuming, especially as the complexity of a game level increases or as the number of active game entities increases. It is beneficial in regard to memory and processor efficiency to simplify the path-finding problem and to reduce or eliminate the reliance on computationally expensive runtime calculations, such as the ubiquitous A* algorithm, to find a solution. The A* algorithm for path-finding can be exhaustive and wasteful, especially when a path is not possible, and even more dangerous when working under constrained memory requirements, such as those found on the current console systems [Stout 2000]. The NavMesh object provides a preprocessed solution to path-finding between any set of triangles on the same NavMesh. This is achieved by storing the edges in a connection map that define the closest connected edge to move from each triangle to every other triangle in a NavMesh object. Runtime computations are reserved for the accuracy of movement from a source point to target point solution, as well as for linked movement between NavMesh objects.

Navigation Mesh Overview

The basic concept behind a navigation mesh is the simplification of movement and static collision within the virtual 3D world. The creation of a navigation mesh as a set of triangles defines the playable movement boundaries in which the AI can navigate. As previously mentioned, the connection between neighbor triangles is stored in the connection map array. By matching the NavMesh

triangles closely to the static geometry of a game world, the navigation mesh can greatly reduce the reliance on collision detection routines, at least as far as nondynamic interactions are concerned. If an arbitrary AI entity can only occupy space on a valid NavMesh triangle and the NavMesh only defines valid space mutually exclusive of static world geometry or terrain information, then the collision with static geometry cannot occur through normal clamped movement on the triangle mesh. Navigation meshes do not have to conform exactly to the visible static geometry of the game world, especially if there are areas in a game level where the designers do not want computer-controlled characters to be permitted.

The scalable path-finding *Iterative Constraint Rays* algorithm uses the connection map in conjunction with the source and destination positions to determine the optimal direction to move to reach the destination. Depending on the CPU time allowance for AI path-finding, multiple successive iterations into the algorithm will provide more accurate results in regards to forward planning against changes in the NavMesh topology. The number of iterations is set depending on the game requirements; existing games at Vicious Cycle Software use a typical value between five and ten iterations to provide sufficient movement results with minimal processor overhead per entity. An example is discussed in the *Iterative Constraint Rays Algorithm* detailed later in this article.

One of the goals when creating an efficient NavMesh object is to keep the triangles as large as possible while maintaining as tight a fit as needed into the game playfield. The path-finding algorithms base the current location of an entity on the current world position combined with the corresponding navigation triangle that contains the entity's position. The collision avoidance solution is simplified for static geometry in that the movement and desired velocities that can be traversed during one game "tick" will never be permitted to move outside the valid bounds of a NavMesh triangle.

All code behind the implementation of the NavMesh object is written in C++ with only single-inheritance to a base object class for name and handle specification. To avoid code bloat and necessary concerns for memory usage, there is no use of templates or the Standard Template Library in the implementation. Triangle information is stored in a structure and is packed based on structure alignment to maximize contiguous memory usage. The base class to the NavMesh object is the same base class to other game-level resources (such as static geometry, dynamic geometry, materials, textures, and so forth). By inheriting these name and handle members, the NavMesh objects can be stored in groups and referenced by handle for comparison when game entities need to maintain a position or refer to a destination.

The precomputed solution to navigation path-finding is performed when the NavMesh triangles are generated and subsequently modified in the game level editor. The data that contains an immediate solution for moving from the current triangle to every other valid triangle on the same NavMesh via one of the three triangle edges is stored in the connection map array. A simple example of an editor session creating a NavMesh object is shown in the Figure 4.2.

By creating multiple NavMesh objects in a game level, the search space for clamped triangles (positions that begin outside the movement bounds of the active NavMeshes) is greatly simplified. If one large NavMesh object were used with no limit on the number of contained triangles, searching for valid points could require time-consuming search functions. During closest-point evaluations, the NavMesh objects can be evaluated using their axis-aligned bounding volumes to determine quickly which of the active NavMeshes contain the queried point. The axis-aligned volumes are generated when the NavMesh object is modified in the game editor and simply consist of the furthest two vertex extents. Additionally, all clamping functions compare the closest NavMesh below the object's

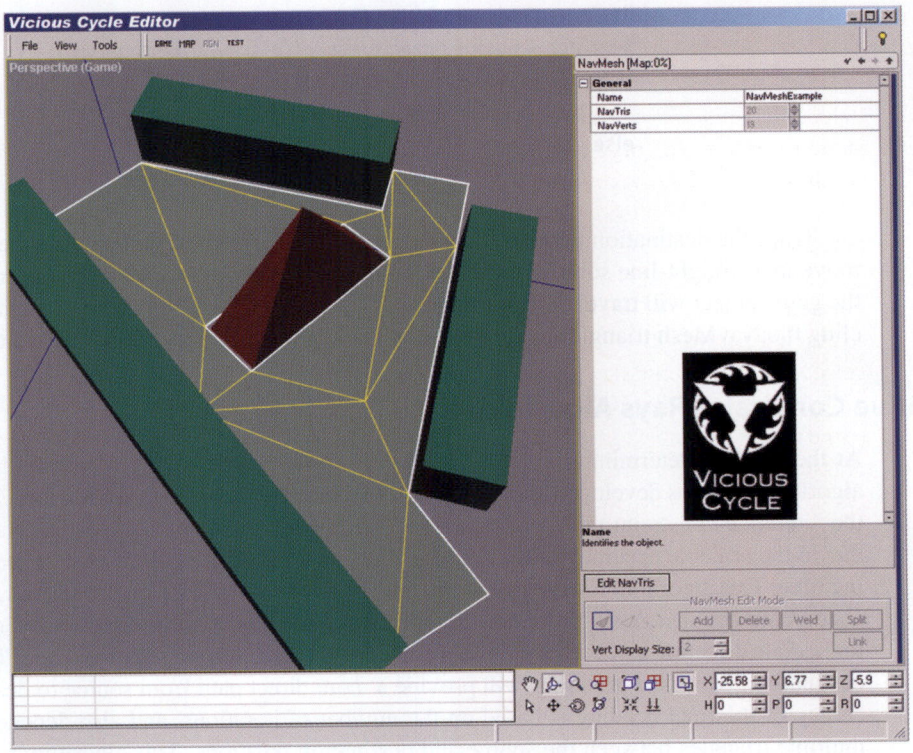


FIGURE 4.2 Simple NavMesh in the Vicious Cycle *Editor*.

current world position; any active NavMesh object that has all vertices above the compared position are culled out of the evaluation.

Path-Finding Solution Overview

All game objects that are to be clamped and bounded by one or more NavMesh objects can take advantage of the connection map information for rapid path-finding solutions. When an object's position is contained "on" or "within" a triangle, the current triangle index can be used as a reference for edge look-up into the connection map. The resulting edge can be used as the intermediate movement point to travel between in order to reach another distinct point on the same NavMesh. The differences between containment "on" and "within" a triangle are discussed further with regard to the collision avoidance solution.

During the initial creation of the set of NavMesh triangles, an array of edge indices that constitute the connection map is allocated. This connection map contains a 3-bit entry for each triangle's connection to every other triangle in the same NavMesh. This connection data is used to determine whether a distinct path exists between the set of all triangles in the mesh. This information is used during the evaluation stage of the AI's path calculation and specifies which edge to travel through to reach the destination triangle, or the next closest triangle in the path. The following algorithm describes the basic path search function:

```
while CurrentTri is not DestinationTri
    NextEdge = Query connection map(CurrentTri, DestinationTri)
    if NextEdge is an invalid edge (not 0, 1, or 2)
        path does not exist - exit and return failure
    else
        CurrentTri = neighbor triangle at edge NextEdge
```

Once the destination triangle index is equal to the current triangle index, the game object can move in a straight-line solution, as every point on the same triangle can be approached. Note that the game object will travel unhindered by static collision due to the creation methodology of modeling the NavMesh triangles exclusive of all static geometry that would cause a collision.

Iterative Constraint Rays Algorithm

At the heart of determining a valid direction to move when solving a path-finding query lies an algorithm that was developed specifically for this implementation of NavMeshes. It is referred to as the "iterative constraint rays" algorithm and significantly extends the existing NavMesh solutions due to the scalable accuracy of the solution that can be generated at the cost of more CPU processing time. This algorithm is designed to run multiple iterations, based on the desired accuracy of look-ahead calculations. A single iteration will always provide a valid direction of movement in the immediate direction toward the neighbor triangle that leads the source object one "hop" closer to the destination. Increased iterations will provide a more direct path from source to destination. The accuracy of the final heading is based on the number of iterations, and also depends on there being multiple triangles between the source and destination positions. This algorithm is used to obtain a relative heading to get from a current position to the target position on the same NavMesh. To navigate between positions on different (linked) NavMeshes, the target point becomes the linked triangle and edge that the NavMesh reports as the current solution to get from the current NavMesh to the destination NavMesh. The iterations are performed to define the range of valid directions from the source position to the target position. Once the target is reached or the maximum number of iterations has expired, the vector from the starting position to the target point (in the case of reaching the target triangle) or last movement edge (in the case of the target triangle not having been reached) is compared against the constraints. If the vector is within the constraints, it is returned as the current solution; otherwise, the closer of the two constraint rays is used to determine the current solution. The initial number of iterations can be passed as a variable from the game-level, based on the amount of time available to the update processing for each active AI entity.

Iterative Constraint Rays Algorithm

Initialization of containment rays:

1. Set `edgeCurrent` equal to the path solution from the current triangle to the destination triangle.
2. If `edgeCurrent` has one or more solid vertices (vertices on a triangle without one or more neighbors), define the initial containment rays as normalized vectors from the starting point to each of the solid vertices. Solid vertices are discussed in the section "NavTri Edges, Solid Edges, and Solid Vertices.

3. If only one vertex is solid on `edgeCurrent`, negate the alternate constraint ray and bias the direction toward `edgeCurrent`. This prevents cosine evaluations of 180-degree angles.
4. If no vertices are solid on the current edge, the containment rays are left empty and this initialization step is deferred until the first solid vertices are encountered in subsequent iterations.

Finding local direction (within the same NavMesh) from source to destination:

1. Verify *sourcePos* and *targetPos* exist on the same NavMesh. Both variables are *NavPos* structures (containing world position, handle to current NavMesh, and source triangle index).
2. If *sourcePos.nTriangle* = *targetPos.nTriangle*, return vector from *sourcePos.vWorldPos* to *targetPos.vWorldPos* since both points exist on the same triangle.
3. Set *nEvaluateTriangle* = *sourcePos.nTriangle*.
4. Repeat while solution not found and iterations exist:
    5. If *nEvaluateTriangle* = *targetPos.nTriangle*, then iterations have reached the target triangle; compare destination point against constraint rays and return bounded solution.
    6. Else, set *edgeCurrent* to move from *nEvaluateTriangle* to *targetPos.nTriangle*.
    7. Evaluate the two vertices on *edgeCurrentTarget*, and if either is solid, update constraint rays.
    8. Set *nEvaluateTriangle* to neighbor triangle on *edgeCurrent*.

The following example NavMesh contains a set of triangles and a source point located on triangle 1 to a destination point that is not on the same triangle. During the first iteration of the path-finding evaluation, the two constraint rays are initialized based on the first edge solution. The NavMesh connection map reports that the connected edge from the source to destination is set to the edge connecting 1 and 2. The constraint rays are initialized with two solid vertices found on the current edge, and the neighbor triangle is evaluated. Because the connected triangle is not the destination triangle, the next edge is evaluated and the constraint rays are clamped to the next set of solid vertices, as shown in Figure 4.3 drawn as two dotted-line red vectors.

During the second iteration, since a solution was not found and the number of iterations has not expired, the current triangle being evaluated (triangle 3) is used to find the next edge solution. The next edge returned by the connection map query is the edge shared between triangles 3 and 4. The constraint rays are updated based on the new edge, and because there is a solid vertex found, the old constraint ray is compared against the new edge. Since the vector from the source point to the solid vertex is within the current constraint rays, the constraint rays are moved "inward" to smaller bounds (see Figure 4.4).

Continuing the evaluation of solid vertices on the connected edges, the constraint rays become closer together, up to a maximum tolerance. If the maximum number of iterations is used, the destination triangle is reached, or if the maximum tolerance on the constraint rays is reached, the evaluation is terminated. In the previous example, at five iterations the constraint rays have moved inward a significant amount and the destination vector is compared. Because the vector is outside the constraint rays, the resulting solution is bounded to remain within the constraints (see Figure 4.5).

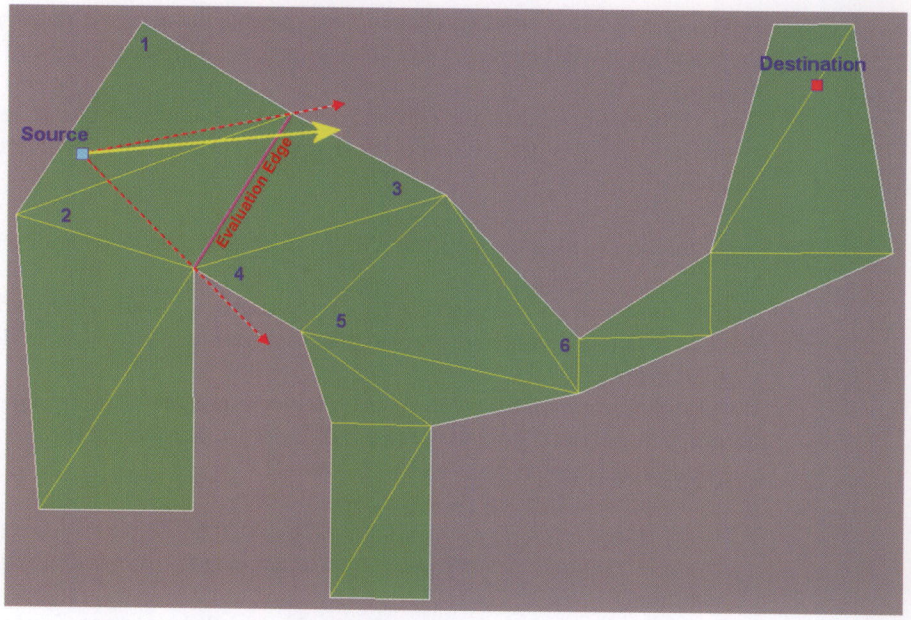


FIGURE 4.3 Single iteration for path-finding.

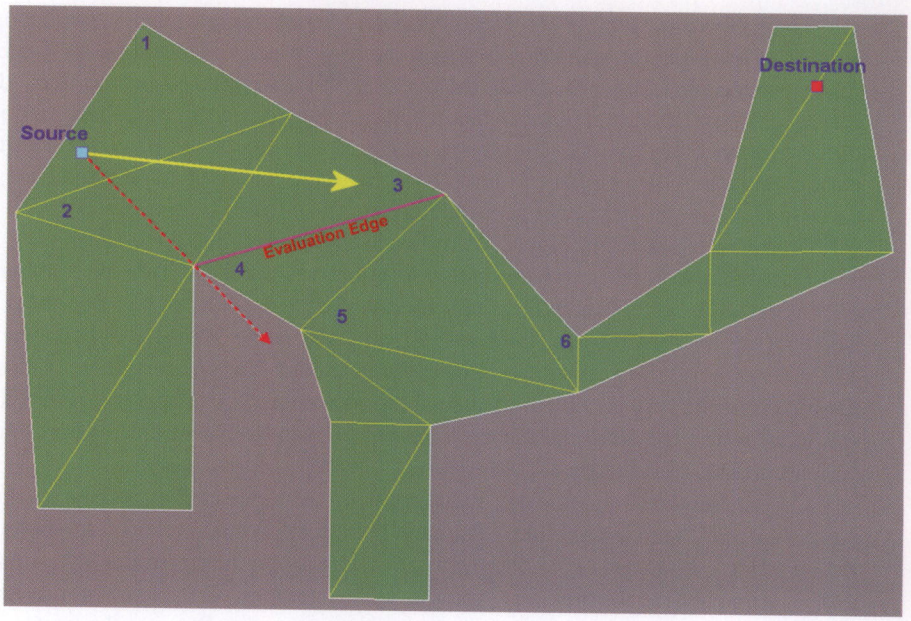


FIGURE 4.4 Two iterations for path-finding.

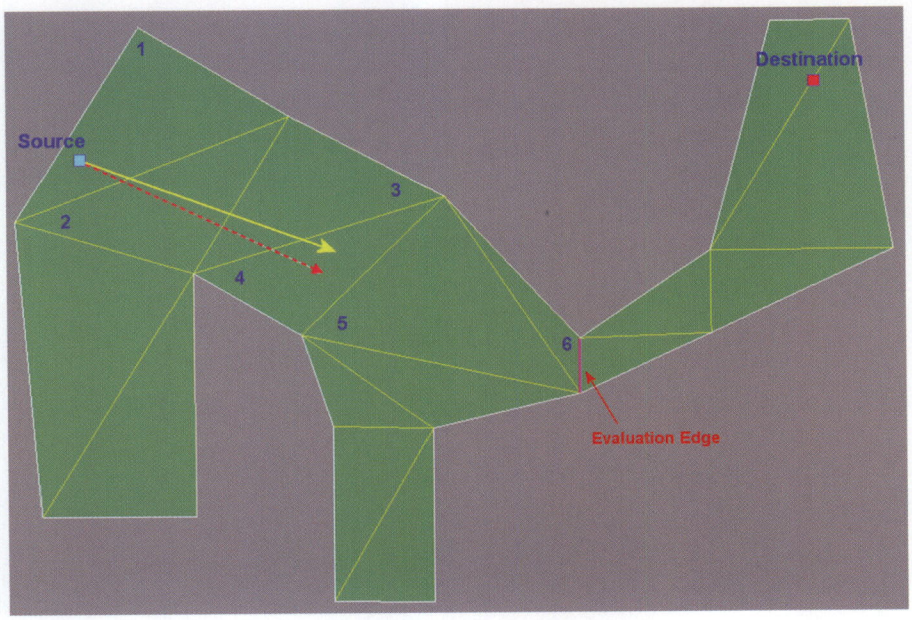


FIGURE 4.5 Five iterations for path-finding.

Determining containment between the constraints involves evaluating the dot product of vectors and using the resulting dot product as a cosine evaluation to determine the smaller of the two angles. Figures 4.3 through 4.5 show dotted red lines to indicate the constraint rays. The solid yellow vector indicates the source to destination vector, clamped to exist within the constraint rays. When a constraint ray is used as a solution, it is biased slightly toward the opposite constraint ray to prevent movement along a solid boundary when the starting position is very close to a solid triangle edge. Each iteration of the algorithm provides a valid direction for movement. Increased iterations provide for more direct paths from the source-to-destination position, with solid edge collisions being considered and compensated against. If the collision radius of the source object is provided to the path-finding algorithm, the constraint rays are biased away from solid vertices by the radius. This aids in the avoidance of edge collisions (see Figure 4.6).

Collision Avoidance

To aid in static collision avoidance for AI-controlled entities, the NavMesh can provide guaranteed valid positioning that will never collide against static world geometry. The first requirement to this solution is that the NavMesh is modeled in the game editor appropriately; any overlap between a NavMesh object and game geometry will result in either collision with the static geometry, or more likely, interpenetration with the visible scene. Positions in world space are queried against the current NavMesh object—if previously cached—or against the set of all NavMesh objects to determine the exact container triangle or the closest NavMesh to the starting point. Cached positions may be stored when a game object has been examined for its NavMesh position (handle and triangle index) and are invalidated whenever movement is detected on that object. By caching positions,

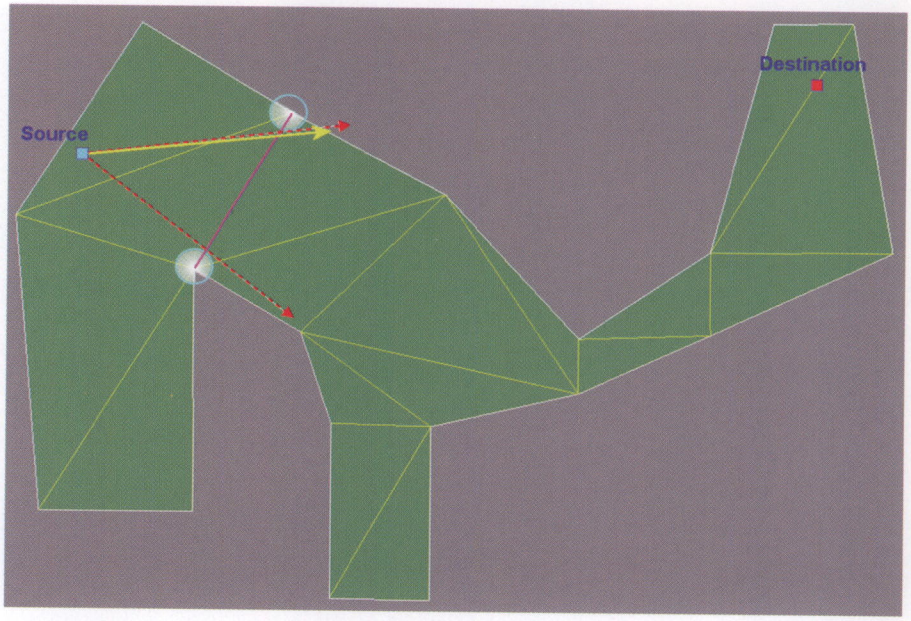


FIGURE 4.6 Using radii versus point evaluations.

multiple requests for NavMesh information during the same game update loop will require only one call to the position validation function. A collision radius can also be passed into the NavMesh query and path-finding functions to allow for treatment of the game object as a sphere rather than a point. In the case of a position that starts outside all NavMesh objects, the option to clamp to the closest NavMesh allows the game a guaranteed valid starting position on a NavMesh; optionally, a warning may be reported.

As mentioned previously in the path-finding overview, there are two types of clamping that may occur when updating a game object's position relative to the current NavMesh triangle index. Containment "on" a triangle means that the position of the game object is clamped to remain on the plane of the current triangle. The position is updated during each tick cycle of the game and can be verified if any movement is recorded for a specific update; otherwise, the position is maintained as being valid and the current triangle index does not need verification. Containment "within" a triangle simply keeps the horizontal bounds (two axes, typically X,Y position) within the volume defined by each triangle; vertical bounds are infinite. Using positions "within" the NavMesh allows for vertical variance in object movement; for example, in the case of a flying object that needs to change altitude during movement.

All movement that is applied to a NavMesh-bound object via animation-based or physics-based movement must be validated against the object's current NavMesh. The game object maintains a current NavMesh position that contains a current NavMesh object handle, the current triangle in the NavMesh, and the corresponding world-space 3D position. By validating all motion that occurs from any point within a valid starting point on a NavMesh, the movement and resulting position can be as-

sured as valid. Movement velocities are compared against the valid set of triangles in the current and linked NavMesh objects, and invalid movement is clamped at triangle edges that do not have attached neighbors. These edges are referred to as *solid* because they define the boundaries of movement and will not permit objects to pass. Any vertex that defines one or more solid edges in the NavMesh triangle array is marked in the NavMesh triangle data and will be used for comparison during the Iterative Constraint Rays algorithm.

Organization of Navigation Mesh Data

The following set of rules applies to the creation of a NavMesh object and provides the basis for efficient memory usage:

1. Maximum triangle limit of 255 to allow single-byte indexing. Entry 255 (hex FF) is reserved as an invalid triangle index, leaving valid indices in the range of 0 to 254.
2. Maximum NavMesh vertex limit of 3 * MaxTris – 2, since every triangle must be connected to at least one neighbor.
3. All vertices are stored in fixed-point format, meaning they are stored as 3 signed shorts and converted to or from world space positions by the multiplication of a scaling factor to each coordinate axis. This is described later in this section.
4. For the set of all triangles in a NavMesh, there exists a path from the current triangle to every other triangle such that the number of "hops" is less than or equal to the number of triangles minus one. Each connection may be stored as an iterative solution such that a single lookup into the connection map will reveal the current movement edge. The current solution provides the correct edge that approaches the destination triangle, but the cumulative solution is not available without multiple iterations through traversal between neighbor triangles.
5. Each triangle in a NavMesh can have a maximum of one link to another NavMesh object. The single link per triangle limit is to allow efficient storage within the triangle structure; otherwise, an additional two indices would be required to store potentially one, two, or three link edges.

Console systems' memory and performance requirements force a necessary efficiency restriction to all memory and processing usage, especially in regard to systems that are updated continuously or multiple times each game update cycle. To help maintain compact and efficient use of memory, each NavMesh may contain a maximum of 255 triangles and 763 distinct vertices as described previously. This differs from other implementations of NavMesh objects where the goal is a tight-fit to the game playing field [Tozour 2002]. This implementation focuses on maximizing the information in the NavMesh object while using a minimal amount of storage and processor time. The storage type for the triangle index is an unsigned byte, and the last valid position index (value 0xFF) is reserved as an invalid placeholder. An array of valid triangles is generated in the game editor, and at the point of serialization, unused and deleted triangles and corresponding ver-

tices are dropped from the list and only the exact number of required triangles is stored. The structure for a single triangle is as follows:

```
struct NavTri
{
   uchar nVert[3];          // Index into array of vertices
   uchar nNeighborTri[3];   // Index of connected triangles
   uchar nFlags;  // Information flags; see below
   uchar nLink;   // Index into array of NavMesh links
};
```

The resulting size of the `NavTri` structure is 8 bytes that can be optimal for many compiler settings where structure alignment is set to 8 bytes or a multiple thereof. Each triangle contains the `nFlags` byte that contains a bit-field for flags that determine if a referenced vertex is "solid" and if at most one of the edges (neighbor triangles) is actually a link to another NavMesh. In the case of a linked flag, the `nLink` field is set to a valid index into an array of links stored within the NavMesh object itself. The `nVert` indices are populated at the triangle mesh creation time in the editor. The `nNeighborTri` indices are computed during the connection-map generation algorithm, part of the preprocessing step that occurs during NavMesh creation in the game editor. These neighbor indices are initialized to the invalid triangle index (0xFF) and correspond to solid edges. Any movement evaluation that puts an entity's position beyond these edges will be clamped at the first solid edge that is intersected. When a NavMesh is linked to a neighbor NavMesh object, the corresponding triangles are set to contain an index into a list of valid links for the entire NavMesh object. The NavMesh link array contains a list of handles that point to other NavMesh objects and is stored in a resizable, dynamic array to allow for an unlimited number of connections between the set of all NavMesh objects in the current game level.

Connection Map Generation

The connection map structure contains edge information for every triangle in the NavMesh. The values placed in the map provide an edge solution to move from any one triangle in the mesh to any other triangle in the same mesh. The amount of memory required to store this structure is calculated based on the number of triangles in the NavMesh, and is equivalent to 3 bits per triangle times the number of triangles in our mesh.

```
int eNavMesh::CalcConnectionMapSize( int nTris ) const
{
   int nRequiredBits = 2*nTris*nTris;

   // Round to the closest byte
   if( nRequiredBits%8 > 0 )
       nRequiredBits += 8-(nRequiredBits%8);

   // Return the required number of bytes to allocate
   return nRequiredBits >> 3;
}
```

When the number of triangles changes in the NavMesh, the connection map is destroyed and reallocated. To create the actual edge connections in the map, each triangle is visited and a best distance is calculated and temporarily stored from the center point of each triangle. The temporary distances are used for subsequent iterations of the edge calculation algorithm. All edge connections are computed at the time of editing, and no modifications are permitted once the NavMesh is in use in the game.

Creation of Navigation Mesh Vertices and Triangles

As previously mentioned, the vertex format for the NavMesh consists of a fixed-point signed short array for each of the three coordinate axes, as follows:

```
struct NavVert
{
   short v[3];
   static const float s_fNavVertScale;
   static const float s_fNavVertInverseScale;
}
```

Storing vertices in this format versus using three floating-point values for the coordinate point saves 6 bytes per vertex:

```
struct Vec3
{
   float fV[3];
};

sizeof( Vec3 ) = 12 bytes
sizeof( NavVert ) = 6 bytes
```

The `s_fNavVertScale` is initialized to the maximum span that a NavMesh object may contain. The `s_fNavVertInverseScale` is also initialized at compile time to eliminate the need for a floating-point division when converting back into a normal floating-point vector in world space. For example, setting a maximum span to 256 meters results in an inverse scale of 0.00391, or approximately 4mm precision for all stored values. Increased precision at the cost of reduced spans may be defined, with typical values being 512, 256, 128, or 64 meters depending on the size and scale requirements of a game level.

The access functions to set (convert to short) and get (convert to float) a vertex are as follows:

```
void NavVert::Set( const Vec3& fXYZ )
{
   v[0] = ( short ) ( fXYZ[0] * s_fNavVertScale );
   v[1] = ( short ) ( fXYZ[1] * s_fNavVertScale );
   v[2] = ( short ) ( fXYZ[2] * s_fNavVertScale );
};
```

```
const Vec3& NavVert::Get( ) const
{
   return Vec3(  s_fNavVertInverseScale * v[0],
         s_fNavVertInverseScale * v[1],
         s_fNavVertInverseScale * v[2] );
}
```

At Vicious Cycle Software, our process for the creation of NavMesh objects is done within our game editor. Previous versions incorporated an exporter written for 3ds max™ that would provide the triangle mesh for use in the game editor. The problem would arise, however, when any new static geometry was placed or moved after the game level was initially exported. This would require subsequent changes to the NavMesh, which meant reloading the level in 3ds max™ and modifications of the triangles and vertices, additional exports, and so on until the level was completed. The current tools allow for direct editor access to the NavMesh triangles and vertices, so level designers and artists can make slight adjustments to the level without having to go through the reload and export steps previously mentioned. An initial NavMesh object consists of simply one triangle, with additional triangles being added to a solid edge of any current triangle in the NavMesh. Current vertices can be moved or combined with neighbor triangle objects to eliminate triangles as needed, and edges may be split to form tessellated triangles in complicated regions.

NavTri Edges, Solid Edges, and Solid Vertices

All triangles in a NavMesh maintain a list of three neighbor triangles that correspond to the adjacent edge of the triangle. Solid edges are formed when a NavMesh triangle does not have a neighbor triangle on a specific edge. These edges form the boundary that movement tests and clamping are compared against. Solid vertices are marked in the `NavTri` structure if any of the three vertices defining the triangle are used to form a solid edge. Edges are used to determine navigation direction when moving between connected triangles in solving a path from point to point on the NavMesh as well as clamping movement when the distance traveled in one frame intersects a solid edge. For example, in Figure 4.7:

- Solid edges are drawn in black. Nonsolid edges are drawn in green.
- Solid vertices are drawn in black. Nonsolid vertices are drawn in orange. In this example, there is only one nonsolid vertex.
- Triangle 1 contains two solid edges and one nonsolid edge formed with neighbor triangle 2.
- Triangles 3 and 4 contain three nonsolid edges, but the vertex that is shared between them (the black vertex that is also shared with triangles 2 and 7) is marked as a solid vertex. Even though triangles 3 and 4 have no solid edges, triangles 2 and 7 have a solid edge that uses the common vertex. This solid vertex will be used in the evaluation of constraint rays during path-finding.
- Triangles 3, 4, 5, and 6 share a common vertex (in orange), and the common vertex is nonsolid since there are no solid edges that have that vertex as one of the two endpoints.

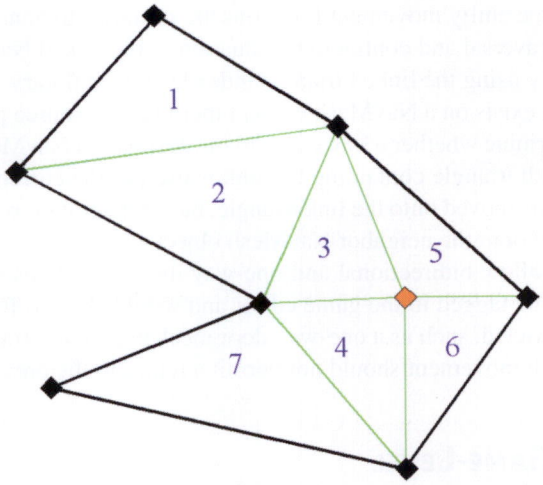


FIGURE 4.7 NavMesh solid edges and solid vertices.

Navigation Mesh Links

Due to the limit on the maximum number of triangles per NavMesh, as well as the dynamic nature of a game level (such as doors, bridges, trap floors, and so forth), the concept of linking multiple NavMesh objects together becomes necessary. Each NavMesh contains a local resizable array of connected NavMesh object handles. The only restriction is that any triangle in a NavMesh can have a maximum of one edge shared as a link. In Figure 4.8, the double-red edges are marked as links between the NavMesh objects. The corresponding triangles in each of the two NavMesh objects will contain a link flag and reference an index that stores the handle to the linked NavMesh object.

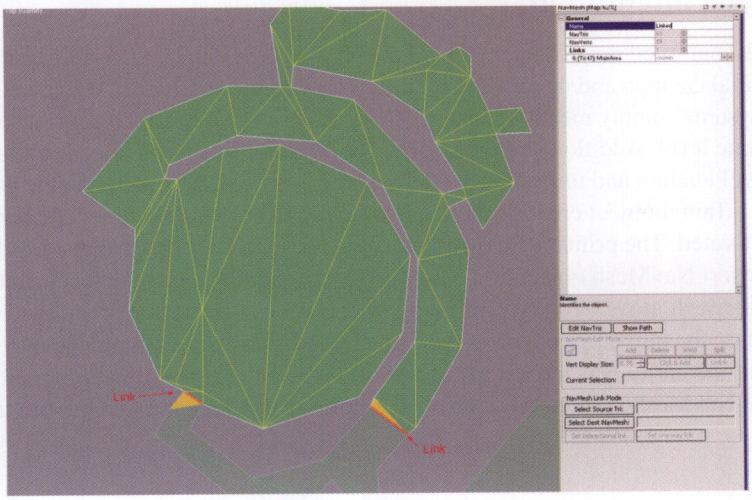


FIGURE 4.8 Linked NavMesh objects.

The in-game entity movement functions are designed to handle link edges as being nonsolid, resulting in a traversal and continuing evaluation in the linked NavMesh object. Path-finding also handles links by using the linked triangle index as the temporary search solution. When the destination position exists on a NavMesh object other than the source position, the current NavMesh is queried to determine whether a link exists to the destination NavMesh handle. If a link is found, the source NavMesh triangle containing the link is used as the current destination position. Once the source object has moved onto the link triangle, the next iteration will result in movement across the linked edge and onto the neighbor NavMesh object.

Links can allow bidirectional and one-way movement between the connected NavMesh objects. This can be tagged in the game editor and would be used for areas where motion and path-finding are restricted, such as a one-way door mechanic or as a transition from a higher area in the level where path movement should not permit a return to the previous NavMesh.

Algorithms and Game-Level Support Functions

The creation of a NavMesh object in the game editor allows for the connection map to be generated beforehand, as well as compression of data if required. All triangle mesh and vertex arrays are sized to contain the exact number of triangles since in-game modification is not allowed—or necessary—once a game level begins. Three main routines manage in-game NavMesh queries: path-finding within a single NavMesh object (source and destination triangles are on the same mesh), path-finding between multiple NavMesh objects, and movement clamping during a game entity's update loop. Movement clamping is handled within the entity update loop, described in the next section. By maintaining the fundamental requirement that all NavMesh objects have a defined solution between the set of all triangles in the same mesh, there always exists a solution between points that lie within the same NavMesh object.

Game Object Integration

AI-controlled entities interact with the NavMesh system at three different phases of their existence: initial creation and/or activation, during movement, and during AI update. The notion of a NavMesh "system" simply means a managed array of active NavMesh objects that are in effect for the current game level. Addition and removal of NavMesh objects is handled in the system based on the game level loading and unloading sections of the playfield, depending on the game requirements.

Initial object creation and activation are handled when the game level is started and the AI is activated. The primary NavMesh position is established by casting a ray downward and finding the closest NavMesh object. If a NavMesh is not found, then the object is not considered bound by the NavMesh movement; typically, this will invoke an error that is reported to the level designer who set up the initial placement in the game editor. This verification stage may also be called at any time that the entity was moved off of a NavMesh and returned to within the boundaries and control of the NavMesh system. AI-controlled entities can be toggled in regard to the bounds checks against

the NavMesh objects. This can be useful when movement is requested but bounded constraints are not required, such as during in-game animation sequences where pre-scripted animation movement is controlling the entities.

When movement is processed, either in response to normal animation-based offsets or a physics movement update, the previous position and desired movement position are compared and the offset during the current update cycle is calculated. This resulting movement is validated with the current set of active NavMesh objects to confirm that movement remains within the defined triangle mesh. Any collision to solid edges that occurs is returned to the entity movement function and the final position is clamped to within the allowed constraints. If the object is treated as a point, it is clamped to the closest encountered solid edge. If the object contains a radius of collision, the offset is added to the clamped position to maintain a set distance from the solid edge.

AI processing evaluates the current target object or movement point and validates that a heading change is needed. If movement toward or relative to the target position is required, the NavMesh system is queried for a valid heading to orient against. In the movement and AI update phases of the update process described previously, there is an assumption that the entity's current position is defined and known to be on a valid NavMesh triangle. This initial triangle index and NavMesh handle is maintained in the following pseudo-position structure, stored within every game entity that is required to be bound within the active NavMesh objects:

```
struct NavPos
{
   Vec3          vWorldPosition;
   NavMeshHandle hNavMeshObject;
   NavTriIndex   nNavTri;
}
```

The `NavMeshHandle` represents an abstract index into a managed array of NavMesh objects in the NavMesh system. The handle may be dereferenced to obtain the actual NavMesh triangle data. The `NavTriIndex` is an unsigned single-byte value representing the index of the NavTri where the `vWorldPosition` exists. As the entity moves or any forces are applied to the object, the entity's `NavPos` is updated and clamped so that it will always remain valid for the current game update cycle.

An error condition is reported if the entity is found outside all valid NavMesh triangles. Initial position clamping is supported to move invalid world positions to be contained within at least one `NavPos` at game startup. When an entity is reactivated or detached from a predefined scripted path (e.g., during a cinematic event), the initial evaluation and position clamping is called to confirm a new starting `NavPos`.

During a normal game update cycle, a number of messages can be communicated back to the game entity's state machine to report NavMesh information or warnings. These messages include events for solid edge collision, arrival at destination position within a configured tolerance, and lost or failed path solution resulting from a link that was deactivated or a destination object that was destroyed or deactivated.

USAGE EXAMPLE

Spy vs. Spy requires the AI-controlled entities to follow the same set of rules and goals as the competing player characters. The use of NavMeshes allows the AI to have the same path-finding knowledge as the human competitor, and since the human player is always able to see what the AI is doing, visible movement "cheating" is not allowed. All of the game levels use multiple NavMeshes to segment the allowed movement and prevent certain areas from being entered without following the prerequisite paths, which is achieved by using the one-way links in key areas. It was during the development of the initial *Spy vs. Spy* levels that the notion of a one-way link was created to prevent the computer player from moving into a restricted room via an "exit only" door. (See Figure 4.9.)

FIGURE 4.9 Linked NavMeshes for a typical game level.

In *Robotech: Invasion*, the game encounters are managed and contained as the player moves through the levels. The game typically takes place in an exterior terrain-based setting, and the NavMesh objects are placed to contain AI creatures within the defined encounters. This differs from the usage in the previous *Spy vs. Spy* example in that the NavMesh objects are used to contain entities to regions of the terrain rather than for navigation. The NavMesh objects are typically made up of large triangles that span large areas of the playing field and prevent entities from moving or pursuing the player characters. This is used to the advantage of the level designers in the game, as progression through the vast terrain requires game encounters to be managed through activation and deactivation; knowing that the AI entities in a set encounter will never move outside of their boundaries aids in this task. (See Figure 4.10.)

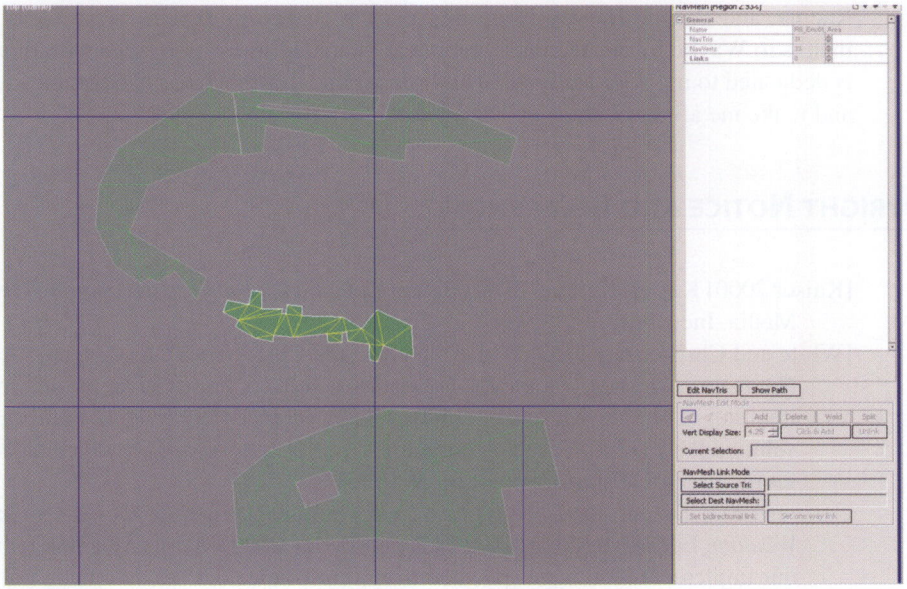


FIGURE 4.10 Example of NavMeshes created to contain encounters.

CURRENT LIMITATIONS AND FUTURE ENHANCEMENTS

Static object collision can be avoided due to the mutually exclusive placement of NavMesh objects, but dynamic object collision must still be processed using alternative solutions. Path–finding for points between different NavMesh objects always locates the closest edge link from the current triangle to the triangle containing the link edge. This will generally provide an acceptable path between triangles on different NavMesh objects, but for NavMeshes with multiple links to the same neighbor NavMesh, manual reassignment of closest links is required to use the additional links.

Linked NavMesh objects are currently searched iteratively based on triangle center positions when determining a closest neighbor. Future work includes grouping sets of triangles in a NavMesh and marking them with different closest links to neighbor NavMeshes rather than relying on a center-to-center determination during the pre-computed connection map calculation.

ACKNOWLEDGMENTS

Many thanks to all of the gifted and talented individuals at Vicious Cycle Software, for without their inspiration and dedication behind the tools and technology, creative ideas, and current products in development, this article would not have been possible. Thanks to Eric Peterson, Marc

Racine, and Wayne Harvey for giving me the chance to express these ideas, and very special thanks to Wayne for his technical appraisal of and editorial assistance with this work. This paper is dedicated to my wife Kelly-May and son Archie, for they are my constant source of inspiration and make me all that I am.

Copyright Notice and References

[Kaiser 2000] Kaiser, Kevin, "3D Collision Detection," *Game Programming Gems*, Charles River Media, Inc., 2000.

[White and Christensen 2002] White, Stephen and Christopher Christensen, "A Fast Approach to Navigation Meshes," *Game Programming Gems 3*, Charles River Media, Inc., 2002. The first iteration of our NavMesh solution was based on many of the ideas in this article. The current solution has evolved based on our games' needs, but we still provide thanks to the authors for giving us a great starting point of inspiration.

[Tozour 2002] Tozour, Paul, "Building a Near-Optimal Navigation Mesh," *AI Game Programming Wisdom*, Ed. Steve Rabin, Charles River Media, 2002. Many of the link requirement ideas for our implementation were spawned from this article, and the automatic polygonal mesh generation functions are intriguing for complex scenes where automation is helpful to the production pipeline.

[Stout 2000] Stout, Bryan, "The Basics of A* for Path Planning," *Game Programming Gems,* Charles River Media, Inc., 2000. This article is a great straightforward description into the details of the workings of the A* algorithm and how it can be applied to games.

GAME DEVELOPMENT AND SOCIAL SCIENCE

Edward Castronova
Cal State Fullerton
ecastronova@Exchange.fullerton.edu

POSITIONING THE THEORY OF GAME DEVELOPMENT

The founding of an academic journal on game development is an assertion that game development has become an important human activity, and that there are some general theoretical principles involved. This forces a question: if game development deserves to be studied in the academy, and it certainly does, we still have to ask, "Where?"

When universities were founded, game development would probably have been claimed exclusively by the fine arts; it would have been placed with dance and theater as another way to create aesthetic experience by combining raw materials and patterned human action. After Freud, however, scholars of the mind would have begun to claim some ownership, with the inherent interactivity of games being seen as their key f-eature, a tool of self-discovery and, of course, fun. With computers, the sciences get involved; the praxis of delivering interactive content involving realistic virtual environments comes to require a deep theoretical knowledge of both the natural sciences as well as information technology. Thus, in the context of contemporary academe, Game Development can be said to involve academic departments of Literature, Dance, Theater, Visual Arts, Film, Biology, Neurology, Psychiatry, Psychology, Computer Science, Mathematics, Physics, Chemistry, and Communications.

This is quite a list, but it might actually cover most of the theoretical ideas that are useful for single-player or small-group games. However, in the last few years, the scope of game development has expanded even further and developers now write games for thousands or millions of players at a time. The core theories of social science have to be considered as well. Therefore, we might add the departments of Economics, Government, Sociology, Anthropology, History, Business, and Public Policy to the list.

Evidently, abstract thinking about game development requires the mind of Erasmus—or the aspiration to such a mind, anyway.

On the Divine Right of Designers

However, do game developers really need to know the social sciences? There are no social structures, no cultural artifacts, and no markets in a game world other than the ones intended by the designer. Or, if there are, it is matter of game design to get rid of them. It might not be easy to do so, but it is always possible, in principle. The designer can even pull the plug. Poof, the offending social structure is gone. *l'Etat, c'est moi.*

Louis XIV (the Sun King, the Absolute Monarch par exemplar) believed in principle that he was France and France was he, and everything else in the country was ephemeral. In principle, it makes sense: all lands owned by the king, all subsistence coming from the land, all people obtaining rights to land and hence subsistence through a vassalage to the king, the king therefore owing nothing to anyone but God. Louis XVI, the great great great great grandson of Louis XIV, was pressured into holding elections for the Estate-General, setting in motion a revolution that deposed him. He was decapitated by the mob on January 21, 1793. As Louis XVI learned, the human mind, even that of an absolute monarch, is built in such a way that it yields to the wishes of others. Monarchs abdicate, dictators call for elections, and presidents resign. The State is not a person, it is the people. Through politics, the people can change almost anything. Indeed, we have seen within our own lifetimes that the most monolithic power structures will eventually collapse if unsupported by the public for long enough.

The public is therefore to be respected.

Those who try to sell games understand this all too well, of course. Marketing departments always want the game to be designed to give the public exactly what it wants. Developers respond, sensibly, that having public satisfaction as the principle objective does not make the design task any easier. Giving people what they want is easy—if you know what that thing is. Even the people themselves, on both an individual and collective level, might have incredible difficulty articulating exactly what they want. A designer who chooses to satisfy every player's whim, like a king who seeks the favor of the people, hasn't ended his problems at all—he's only just begun them.

The Design Uses of Social Science

What kind of problems are these? In the end, for both king and developer, the problems reduce to issues of public policy. In some formulations, public policy science is referred to explicitly as an effort to design the "rules of the game;" the only difference in a game development context is that one removes the quotation marks. Moreover, we know, from Earth's experience with public policy, that good policymaking requires the mastery of all kinds of difficult tasks. The interests of the people have to be aggregated in some fashion, and observed; design decisions have to be proposed; the reactions of the community to those decisions have to be estimated; and the value of the predicted outcome has to be measured. These jobs are hard to do well, and there's no question that public policy works better for everyone—the leaders as well as the people—when it is informed at each stage by general principles of social science.

Understanding interest aggregation systems, for example, is within the domain of political science. That is because interest aggregation is the key service provided by contemporary political and media systems. Parties and lobbying groups and newspapers all provide ways for those in charge

to see where desires are greatest. Absent these structures, rulers are flying blind, which is why governments of controlled societies consistently crash. Rulers need information, and it is too costly to collect by brute force; therefore, it must be allowed to percolate up from the ground. Political science has worked on these questions for a long time, which makes it a good resource for both rulers and game designers.

As a second example, measuring the likely effects of policy and the public's behavioral reactions falls within the domain of economics and other predictive social sciences (sociology, anthropology). Changing the costs and benefits of actions will push behavior in predictable ways, and it is not enough to make proclamations of broad design effects. "If we increase the power of the spell, more people will use it." Yes. However, society is characterized by equilibrium states, and disrupting society out of an equilibrium, perhaps by changing just one spell, can start a dynamic that heads off into radically different regions. The spell attracts use; players respond by changing equipment; that energizes the market for a certain component; players start a gold rush for the component; the component only exists in a single small wooded area; the server hosting that area keeps crashing; players storm the message boards, demanding more stability of the service. These forces are incredibly complex, of course, but they are the bread and butter of the quantitative social sciences. Again, both rulers and developers would find these fields useful for whatever predictive power they can bring.

In sum, as long as game development involves communities of players, it will need the same expertise that government needs; namely, in figuring out what to do and then how to do it. The reactions and characteristics of human communities will clearly play a role, and understanding these features is what social science is all about. Therefore, increasingly, game development will come to involve not just the human spirit in isolation, but also in community.

The Social Science Uses of Design

Because of the similarity between the metaphorical "rules of the game" in public policy design and the actual rules of a game with many players, social science can be useful for game development. Interestingly, the same similarity of function also makes game development useful for social science. This is a second sharp edge to the game design/social science sword: you might want to use your knowledge of human society to code a world, but once you've done your coding, you can use that world to learn more about the human societies that occupy it. As part of their daily business, game developers routinely take ordinary people and place them in extraordinary situations. If properly structured, game design can also take ordinary people and place them in situations that are not just extraordinary, but extraordinary in a specific, consciously chosen way. Games came be turned into organic experiments in the human condition.

For example, economists have debated for centuries whether inflation has any significant effect on what they call the "real" economy: production, employment, and growth. The idea is that money is only a denominator; perhaps it does not matter at all how many zeroes are on the dollar bill that measures production, production will still be the same. Virtual world designers could build two parallel worlds that differed only in the degree to which the money supply was allowed to accumulate. Would the high inflation world have the same number of items in it, meaning more production? Right now, we don't know. However, it would not be hard to find out, once the technology of synthetic world building is deployed for social science research.

Beyond this experimental use, game design has an even more important role to play in the deeper questions of social science. To return to policy issues: yes, our current understanding of public policy, drawn from many different fields of social science, can readily inform game design. However, game design itself represents a powerful criticism of contemporary public policy debates. Public policy, as stated previously, is generally viewed today as a question of giving the public what it wants. The public's desires are usually structured by media and conversations in terms of problems—there's not enough employment, services are too expensive, schools are too dangerous. Policymakers then naturally conclude that solving these problems is, itself, the objective of good policy.

However, any game designer knows that solving the player's problem is not what the player wants. The player doesn't want an answer, he wants a problem that is fun to solve. Apply this game-design reasoning to public policy, and you come up with quite an interesting perspective: public policy, viewed this way, is not at all about solving the public's problems. No, viewed as game design, public policy is supposed to provide every citizen with a problem environment that is satisfying to her, rather than frustrating. The problems might or might not be solved, but if policy does its job, then confronting them and living with them and trying to tackle them will be as satisfying for the citizen as gaming is for the player.

It is worth saying that this perspective is a radical departure, not to be found anywhere in social science today. In 100 years, perhaps, it might be the only thing that social scientists talk about. Now that game development has taken up the challenge of making games with many players, it has entered territories that social scientists have been exploring for ages. The encounter can be useful for the game developers right away, but the biggest impact will come when the social scientists migrate into the territory of game development. In that happy land, the role of the government will no longer be to mollify the people—but rather to help them thrive.

CHARLES RIVER MEDIA
10 Downer Avenue
Hingham, MA 02043
781-740-0400
781-740-8816 (FAX)
info@charlesriver.com
www.charlesriver.com
www.jogd.com